Preuves de Propriétés de Classes de Programmes par Dérivation Systématique de Jeux de Test

Valérie-Anne Nicolas

N° d'ordre: 2118

THÈSE

présentée devant

l'Université de Rennes 1

pour obtenir le grade de :

Docteur de l'Université de Rennes 1
Mention Informatique

par

Valérie-Anne Nicolas

Équipe d'accueil : IRISA, Institut de Recherche en Informatique et Systèmes Aléatoires
École Doctorale : Sciences Pour l'Ingénieur
Composante universitaire : IFSIC, Institut de Formation Supérieure en Informatique et Communication

Titre de la thèse :

Preuves de propriétés de classes de programmes par dérivation systématique de jeux de test

soutenue le 21 décembre 1998 devant la commission d'examen

M.	Jean-Pierre Banâtre	Président
M.	Simon B. Jones	Rapporteur
M.	Daniel Le Métayer	Directeur de thèse
M.	Olivier Ridoux	Examinateur
M.	Laurent Trilling	Rapporteur

Remerciements

Je remercie tout d'abord les membres de mon jury de thèse :

- Jean-Pierre Banâtre, directeur de l'IRISA et professeur à l'Université de Rennes I, qui m'a fait l'honneur de présider le jury.
- Simon B. Jones, maître de conférences à l'Université de Stirling (Écosse), et Laurent Trilling, professeur à l'Université Joseph Fourier de Grenoble, pour avoir bien voulu accepter la charge de rapporteur. Je les remercie tout particulièrement de l'intérêt qu'ils ont témoigné pour mon sujet de thèse, un peu «exotique» par rapport à leurs centres d'intérêt respectifs, et des discussions enrichissantes que nous avons pu avoir.
- Je remercie très chaleureusement Daniel Le Métayer, directeur de recherche à l'INRIA, qui a dirigé ma thèse, ainsi qu' Olivier Ridoux, chargé de recherche à l'INRIA, qui l'a co-encadrée. Je les remercie en premier lieu de m'avoir proposé ce sujet de thèse, puis de leur confiance, de leur permanente disponibilité, de leur constant soutien et de leurs nombreux conseils pendant toutes ces années passées a l'IRISA.

Je remercie également Romain Thomas, stagiaire dans le projet Lande durant l'été 98, d'avoir participé pour une bonne part à la réalisation de mon prototype.

Pendant ces trois années de thèse, j'ai aussi eu le bonheur d'enseigner dans le cadre du monitorat. Je tiens à remercier ici tous ceux qui ont pu m'aider dans ce domaine, notamment ma tutrice Anne Grazon et Pascale Sébillot.

Je remercie l'ensemble des membres du projet Lande, au sein duquel s'est déroulée ma thèse, pour leur sympathie. Je remercie plus particulièrement Tommy Thorn, Ronan Gaugne et Valérie Gouranton pour leurs encouragements et leur amitié, mention spéciale à Tommy pour son appui inestimable dans la dernière ligne droite.

Je remercie ma famille d'avoir suivi et soutenu toute cette expérience : un grand merci à ma mère Danièle, mon père Gérard et ma petite soeur Marine.

Pour m'avoir permis de me changer les idées quand cela devenait vraiment nécessaire, et pour leur sens de la fête, je remercie enfin tous mes amis : Cécile, Hervé, Frédérique, Éric, Sylvain, Karine, Olivier, François, Alain, Nic, ... et tous les autres !

Table des matières

Table des figures

Introduction

D'un point de vue conceptuel, le processus de développement de logiciels met en jeu trois sortes d'objets : les programmes, les propriétés et les données. Les propriétés peuvent prendre la forme de spécifications (formelles ou informelles), d'oracles, de propriétés de correction partielle, de types, d'annotations... Les données peuvent être sous forme de jeux de test, d'exemples, de traces... Ces trois types d'objets peuvent ainsi être vus comme les trois facettes d'un logiciel qui, idéalement, doivent être liées par des relations de cohérence (cf. Figure 0.1).

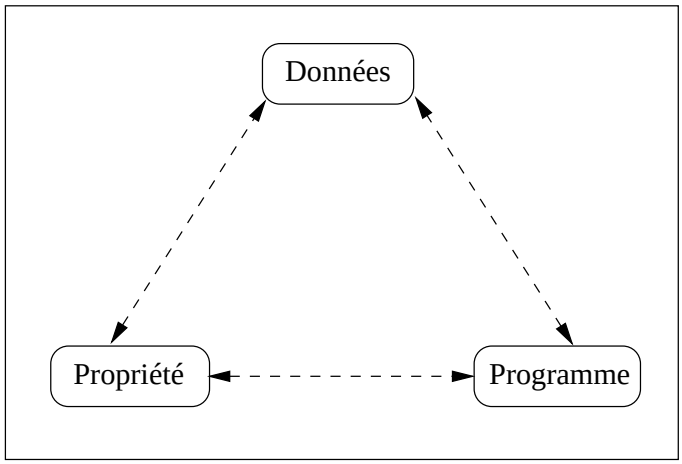


FIG. 0.1 – *La trilogie Programme-Données-Propriété.*

En pratique, même lorsque ces trois facettes interviennent dans le développement d'un logiciel, leurs relations sont rarement établies de manière formelle. Pourtant, nous pensons que cette formalisation peut se révéler d'un grand intérêt tout au long du cycle de vie d'un logiciel. Elle permettrait notamment d'assurer la cohérence d'un programme par rapport à une propriété donnée lors du développement, ou l'adéquation d'un jeu de test par rapport à une propriété et/ou un programme. Lors de la maintenance, cela permettrait aussi une évolution du programme cohérente, en étudiant l'impact de la modification d'une facette sur les autres facettes. Cela serait utile, par exemple, pour le test de non-régression, ou lors de l'ajout de nouvelles fonctionnalités. À l'heure actuelle, dans la majorité des cas, un programme est construit à partir de spécifications, généralement informelles (liste des besoins, cahier des charges), puis testé : il s'agit du

parcours des flèches «Propriété → Programme» et «Programme → Données» dans la figure 0.1. Ainsi, le programme et le jeu de test sont construits selon l'interprétation qu'à le programmeur ou le testeur de la spécification informelle. Il n'y a donc pas de lien formel entre ces trois composants, et par conséquent aucun moyen de les faire évoluer de manière cohérente. Cependant, la production de logiciels critiques (nucléaire, transports...) se fait de plus en plus souvent à partir d'une spécification formelle (de laquelle on dérive le programme), ou de propriétés partielles formelles (devant être vérifiées par le programme). À cela s'ajoute une phase de test qui peut être basée sur la spécification formelle (*test fonctionnel*), ou sur le code du programme (*test structurel*). Ce type de développement illustre bien l'intérêt d'établir de manière formelle les relations existant entre les programmes, les propriétés et les jeux de test. En fait, il se trouve que toutes les relations entre deux des facettes de cette trilogie ont déjà été explorées formellement dans des cadres variés : test fonctionnel, test structurel, dérivation de programmes, apprentissage, preuve de propriétés de programmes, vérification... Cependant, nous pensons [LMNR98] qu'il serait encore plus intéressant d'offrir au programmeur un environnement lui permettant de choisir la manière d'appréhender le développement la mieux adaptée à sa situation. Par exemple, un environnement qui ne nécessiterait pas d'ordre préétabli dans la conception ou l'utilisation des trois facettes, ou qui offrirait des possibilités plus riches comme des méta-flèches «Propriété + Données → Programme» ou «Propriété + Programme → Données»... Dans ce contexte, on peut alors s'intéresser aux relations de cohérence qui lient les trois facettes. En particulier, nous choisissons dans cette thèse d'explorer la relation qui consiste à produire de manière formelle un jeu de test à partir d'un programme et d'une propriété (cf. Figure 0.2). Notre but n'est pas de prouver la correction totale d'un programme, mais plutôt de vérifier certaines propriétés de correction partielle par le test. Peu de travaux ont exploré cette voie qui se révèle être au carrefour des deux approches classiques du test que sont le test fonctionnel et le test structurel.

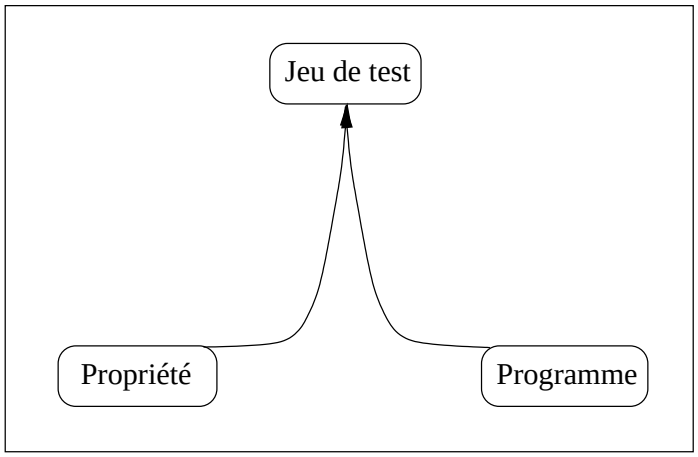


FIG. 0.2 – *Notre approche.*

De manière générale, le test est considéré comme une étape indispensable du déve-

loppement et de la validation d'un logiciel. Bien que très coûteuse (environ 50% du coût total de développement [BEI90]), elle n'apporte aucune garantie formelle sur la qualité du logiciel. Dans la majeure partie des cas, il s'agit en effet d'un processus manuel informel reposant sur l'expérience du testeur, et la relation entre les jeux de test choisis et le programme (dans le cas de test structurel) ou la spécification (dans le cas de test fonctionnel) n'est jamais définie explicitement. Par ailleurs, les travaux théoriques sur le test ne fournissent pas vraiment de base pour des techniques de test utilisables en pratique. Les concepts formels qui en sont issus reposent sur des hypothèses rarement satisfaites en pratique (et difficilement vérifiables), ou conduisent à des jeux de test infinis. Il faut reconnaître que le traitement de ce genre de question en toute généralité se heurte immédiatement à des problèmes d'indécidabilité. Si le langage de spécification est suffisamment riche et que le langage de programmation possède la puissance du calculable, il est hors de question de vérifier qu'un programme satisfait une propriété quelconque avec un jeu de test fini. Nous pensons cependant que cette situation n'est pas sans espoir : elle peut être améliorée en utilisant des formalismes restreints, permettant de faire face à ce problème d'indécidabilité. Cette idée a déjà été explorée dans le domaine de l'apprentissage où, pour rendre décidable le processus de dérivation de programme à partir d'un nombre fini d'exemples, on restreint le domaine des programmes apprenables grâce à un *biais d'apprentissage*. Nous nous inspirons de cette expérience et définissons des hiérarchies de schémas de programmes et de propriétés pour lesquelles nous pouvons construire, de manière automatique, des jeux de test finis complets (c'est à dire garantissant qu'un programme passant le test vérifie la propriété considérée). On peut considérer nos hiérarchies de schémas comme des *biais de test*.

Notre approche permet également d'établir un lien entre le test et la vérification de programmes. Ces deux domaines sont restés plutôt cloisonnés jusqu'à présent (mis à part le cas spécifique des protocoles de communication), alors qu'ils visent un même objectif : augmenter la confiance que l'on peut avoir en un logiciel. Nous pensons qu'explorer les différentes interconnexions existant entre les domaines du test et de la vérification peut être très bénéfique. En effet, par le test, on obtient une information directe sur le comportement du programme pour des exemples particuliers, mais il est nécessaire d'extrapoler ces résultats afin d'identifier le comportement général du programme. Cette extrapolation n'est le plus souvent pas formalisée, ce qui limite la confiance que l'on peut accorder au processus de test. Par exemple, si on considère un polynôme de Lagrange de degré n, on sait qu'il est possible de le caractériser à partir de la connaissance de son comportement sur $n+1$ valeurs. Dans ce cas, on a effectivement une extrapolation valide bien formalisée. Mais il se peut aussi que l'extrapolation soit impossible. C'est ce qui se produit avec les procédures tabulées, pour lesquelles il est nécessaire de tester exhaustivement tous les cas. Le fameux «bug du Pentium» provenait d'une procédure tabulée, et il illustre bien les dangers de l'extrapolation dans le processus de test. À l'inverse, la vérification de programmes ne connaît pas de problème d'extrapolation. En effet, elle utilise des méthodes d'analyse qui donnent des résultats généraux (comme des preuves de propriétés). Cependant, le prix à payer pour cette généralité est que ces méthodes soit sont semi-automatiques et nécessitent alors un utilisateur expert, soit reposent sur des approximations. Intuitivement, notre méthode ressemble beaucoup à

l'extrapolation pour les polynômes de Lagrange : elle concilie la facilité d'utilisation du test avec la robustesse de la vérification (partielle) de programmes.

Le corps de ce document de thèse se décompose en deux parties. Comme notre méthode utilise le code du programme afin de produire des jeux de test, il nous paraît intéressant de commencer par présenter un état de l'art dans le domaine du test structurel (c'est-à-dire basé sur le texte du programme source). Plus précisément, la Partie I est focalisée sur l'étude des différentes techniques structurelles de génération de jeux de test (étape généralement la plus coûteuse du processus de test). Notre méthode faisant également usage d'une propriété à tester, nous avons inclus dans notre survol quelques techniques structurelles «mixtes», au sens où elles combinent l'utilisation d'un programme et d'une spécification (éventuellement partielle). Cette partie se termine par un bilan justifiant les choix de la thèse. Le travail de la thèse proprement dit - une méthode de génération de jeux de test permettant de prouver qu'un programme vérifie une propriété donnée - est détaillé dans la Partie II. Nous concluons ensuite en traçant quelques perspectives d'extension de la méthode.

Première partie

Les techniques structurelles de génération de jeux de test

Chapitre 1

Contexte

Le terme générique de *test* recouvre deux notions complémentaires : celle de test statique et celle de test dynamique. Les méthodes de test statique ne nécessitent pas d'exécuter de programme. Par exemple, les *revues* ou *lectures de code* consistent à faire lire le texte d'un programme par une personne autre que le programmeur, qui lui fait ensuite part de ses critiques. Les *inspections* sont des lectures en groupe suivies d'une réunion où, d'une part le programmeur explique et motive ses choix, d'autre part les lecteurs fournissent leurs interprétations du programme, ainsi que des critiques et conseils éventuels [XMDA+94, GMSB96]. Ce type de test est très coûteux en temps humain mais présente l'avantage d'être applicable et efficace très tôt dans le cycle de développement, en couvrant notamment les erreurs de conception. Cette idée est exploitée avec succès chez IBM, par l'approche dite *de la salle blanche* (*Cleanroom* [CM90]). Néanmoins, il est indispensable de compléter ce type de test par du test dynamique afin de considérer l'exécution réelle, en pratique, du programme. Les méthodes de test dynamique consistent à exécuter un programme sur un jeu de données particulier. Il est donc nécessaire de concevoir ces données au préalable.

Le processus de test dynamique englobe en fait une succession d'activités qui est résumée dans la Figure 1.1 :

- La production des données de test (ou *jeu de test*) selon un *critère de test* donné. Par exemple, pour la méthode que nous proposons dans la Partie II, le critère est de permettre de distinguer deux à deux tous les programmes appartenant à un certaine classe (ou schéma). En toute généralité, cette activité peut s'appuyer sur le programme à tester, sa spécification et une caractérisation de son domaine d'entrée. Cette phase peut être plus ou moins automatisée. Nous utilisons l'expression «génération de jeux de test» pour se référer à une production automatique (ou semi-automatique) de données de test.
- L'instrumentation, qui consiste à introduire des instructions dans le programme source pour récolter de l'information lors de l'exécution de chaque donnée de test (par exemple, le chemin réellement exécuté). Ces informations sont ensuite utilisées lors de l'analyse des résultats (en particulier, elles peuvent servir à mesurer la couverture du code réalisée par le jeu de test dans le cas de test structurel).

- L'application du programme au jeu de test.
- L'analyse des résultats, qui nécessite un oracle (humain ou automatisé) capable de décider si le comportement du programme est satisfaisant. Si elle est disponible, la spécification du programme sert de référence.

À l'issue de l'analyse des résultats, on dispose d'un rapport de test qui indique le succès ou l'échec du test sur chaque donnée de test, et éventuellement des informations supplémentaires comme la couverture du code atteinte.

La «boîte à outils» du testeur comprend des utilitaires destinés à accomplir ces tâches, souvent fastidieuses et coûteuses, de manière plus ou moins automatique. Par exemple, un moniteur de test (*test driver*) peut produire l'exécution du jeu de test de manière complètement automatique à partir d'entrées de test qui lui sont généralement fournies sous forme de table. L'instrumentation peut, elle aussi, se faire automatiquement grâce à des instrumenteurs de code [XMDA+94]. Par contre, l'automatisation de la production des données de test et de l'analyse des résultats est plus difficile et on ne dispose pas de solution générale à ces problèmes. Les seuls outils disponibles sont très peu automatisés. Ils constituent une aide pour l'utilisateur en proposant par exemple des éditeurs pour la saisie des données de test ou différentes analyses du code (construction des graphes de contrôle et de flot de données), et de l'information sur le flot de contrôle ou de données exercés par un jeu de test. Cependant, la production des données de test elle-même demeure la tâche la plus coûteuse. C'est pourquoi nous nous intéressons exclusivement, ici, à l'activité de génération de jeux de test, en montrant dans quelle mesure on peut systématiser cette production, ou même l'automatiser dans certains cas.

Il existe deux grandes familles de méthodes de génération de jeux de test : les méthodes fonctionnelles et les méthodes structurelles [BEI90]. On appelle méthodes fonctionnelles (ou *en boîte noire*) celles qui reposent sur la seule connaissance de la spécification. Ces méthodes peuvent ainsi être utilisées très tôt dans le cycle de développement, avant même de disposer du code du programme. Lorsque le programme est disponible, on peut utiliser les méthodes structurelles de test (ou *en boîte blanche*) qui travaillent à partir du texte du programme. Comme nous l'avons précisé en introduction, nous nous focalisons ici sur la génération de jeux de test basée sur des méthodes structurelles, afin de prendre en compte les trois éléments de notre trilogie (programmes, données et propriétés). Dans ce contexte, nous commençons par introduire plusieurs notions de base pour le domaine du test : tout d'abord la notion de *critère de test*, puis les qualités qui doivent caractériser un tel critère pour engendrer de «bons» jeux de test (Chapitre 2). Nous présentons ensuite un survol des techniques structurelles de génération de jeux de test (Chapitre 3) et nous terminons par un bilan (Chapitre 4).

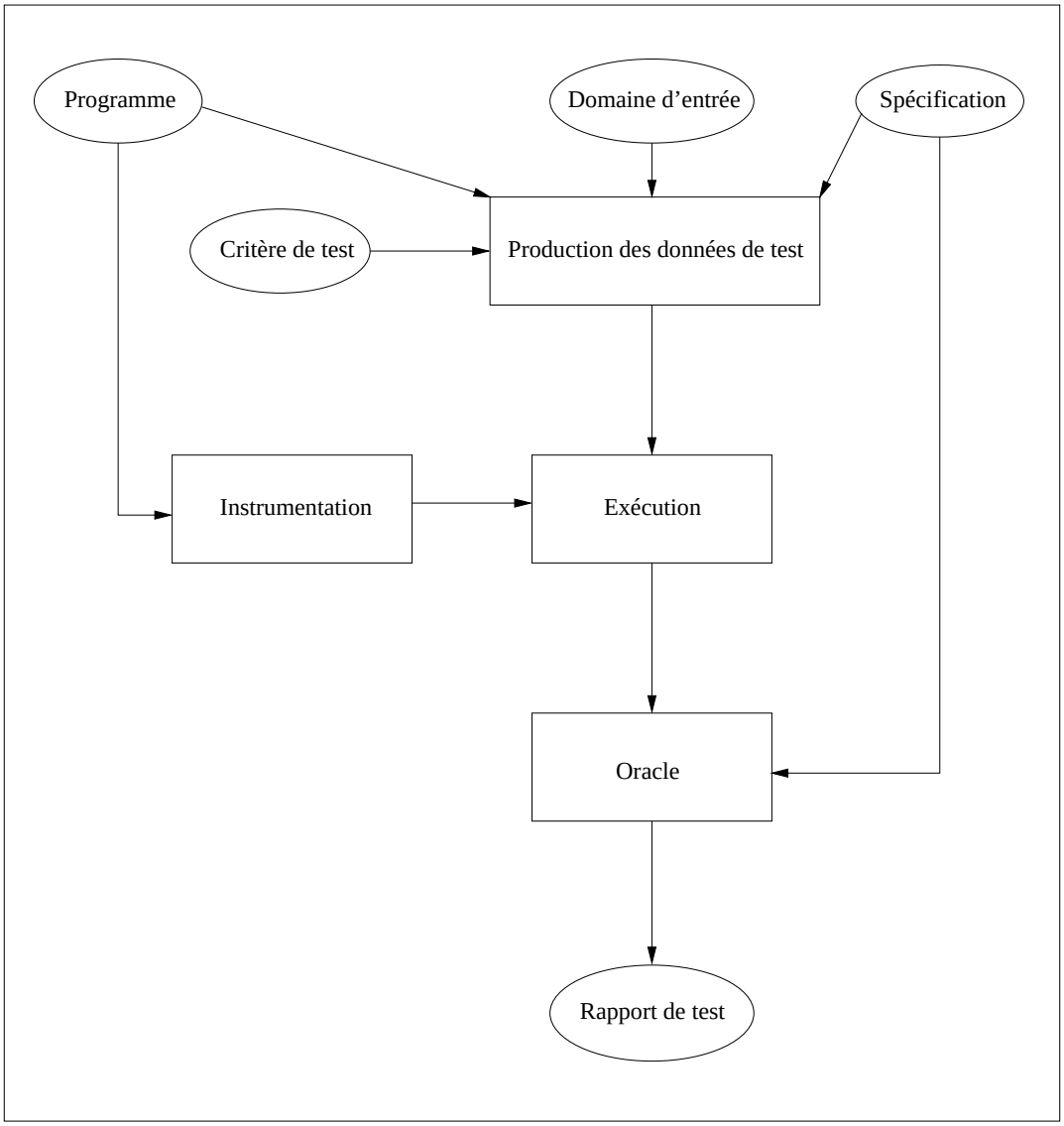


FIG. 1.1 – *Les différentes activités intervenant dans le test dynamique.*

Chapitre 2

Critères de test

Un test idéal (c'est-à-dire équivalent à une preuve) serait le test exhaustif sur le domaine d'entrée du programme. Malheureusement, ce domaine est en général d'une trop grande taille, quand il n'est pas infini, pour que l'on puisse procéder de cette façon. Il faut donc sélectionner des données de test parmi l'ensemble des entrées possibles de manière à obtenir une couverture des erreurs aussi proche que possible de celle obtenue par le test idéal. L'ensemble constitué par ces données de test est appelé jeu de test. La sélection de données de test se fait selon un critère de test. Ce critère est une propriété qui doit être vérifiée par un jeu de test pour que celui-ci soit *adéquat au critère* [GG75] : on peut le décrire comme une relation $\mathcal{C}(P, D, S, T)$, P étant le programme à tester, D son domaine d'entrée, S sa spécification et T un jeu de test. En pratique, on peut utiliser les critères de deux manières :

1. A posteriori : le jeu de test T est alors produit par des moyens quelconques (typiquement, il peut être proposé par le testeur). Son adéquation à un critère donné est obtenue après l'exécution par l'évaluation de la couverture des fautes réalisée (calcul d'une fonction $F(P, D, S, T) = v \in [0, 1]$).
2. A priori : le jeu de test T est généré en tenant compte du critère C (implantation d'une procédure $F(P, D, S) = T$ telle que T est adéquat au critère C).

Les techniques pour utiliser effectivement les critères sont évoquées dans le chapitre suivant. Nous nous focalisons ici sur les définitions elles-mêmes des différents critères et sur les manières de les comparer.

La détermination du critère oriente le test vers la couverture de certains types d'erreurs, et en cela il est associé (souvent implicitement) à certaines hypothèses sur le comportement du programme ou sur la spécification. Intuitivement, tester un programme selon un certain critère équivaut à prouver la correction de ce programme, sous la condition que celui-ci ne puisse contenir que des erreurs du type de celles qui sont détectées par le critère (hypothèse forte sur le programme). Il est important de pouvoir définir formellement ces hypothèses de façon à caractériser les limites d'un test [GAU95]. Il existe deux grandes classes de critères : les critères fonctionnels et les critères structurels. Nous les passons successivement en revue.

Critères fonctionnels :

Le principe des critères fonctionnels est de partitionner le domaine d'entrée du programme en classes d'équivalence. Ce partitionnement peut s'effectuer de deux manières :

- soit selon une loi de distribution donnée : on applique alors une méthode de test statistique,
- soit en identifiant les différents cas qui apparaissent dans la spécification : on se place ainsi dans le cadre du test déterministe des fonctionnalités du programme.

La méthode de test statistique la plus simple est basée sur une loi de distribution uniforme : c'est le *test aléatoire*. Le coût de sa mise en oeuvre est faible mais son efficacité se révèle très irrégulière [TFWC91, NTA81]. Le pouvoir de détection de fautes du test aléatoire est en effet excellent lorsque le domaine d'entrée suit effectivement une distribution uniforme. Mais dans les autres cas, il ne permet de tester correctement ni les fonctionnalités du programme, ni son code. Cette constatation a conduit à déterminer des lois de distribution plus pertinentes en utilisant les informations fournies par la spécification. On obtient alors, de manière régulière, un bon pouvoir de détection de fautes. Mais, si la majorité des fautes détectées le sont très rapidement, il en est d'autres qui échappent généralement au test (car elles correspondent à des cas particuliers et donc improbables selon la distribution considérée). Il paraît ainsi nécessaire de coupler cette méthode avec une méthode de test déterministe permettant d'isoler ces cas improbables.

Dans le cas du test déterministe à partir de spécifications, des travaux ont abouti à des outils permettant de formaliser le processus de test [BGM91, DF93, VA98]. Cependant, dans la pratique, le testeur ne dispose généralement que de spécifications informelles et le processus repose essentiellement sur son expérience. Cette analyse par partitionnement est souvent complétée par l'étude des *conditions aux limites*. Pour cela, on s'intéresse aux valeurs limites des domaines d'entrée et de sortie du programme, ainsi qu'à certaines valeurs particulières du point de vue mathématique ou informatique (par exemple, la valeur zéro et les valeurs proches de zéro). L'expérience montre en effet que la probabilité d'erreur est plus élevée pour ce type de valeurs.

Critères structurels :

Les critères structurels prennent en compte exclusivement les informations issues du code du programme testé. La majorité de ces critères sont basés sur l'étude des graphes de contrôle et de flot de données. D'autres, comme la technique des mutants, s'appliquent aux instructions du programme prises individuellement. On peut aussi classer dans cette famille les critères statistiques lorsque la distribution de probabilité qui les caractérise est établie à partir du code du programme [WAE93]. Nous nous limitons ici à la présentation des critères structurels déterministes.

Dans le reste de ce document, nous nous focalisons sur la génération de jeux de test à partir du code du programme ; nous ne nous étendons donc pas sur les critères fonctionnels. Nous présentons maintenant de manière plus détaillée les critères structurels déterministes. Nous décrivons d'abord les critères structurels basés sur le flot

de contrôle (Partie 2.1), puis les critères structurels basés sur le flot de données (Partie 2.2), et les critères structurels basés sur les fautes (Partie 2.3). Pour conclure, nous évoquons les différentes manières de comparer les critères de test dans la Partie 2.4.

2.1 Critères structurels basés sur le flot de contrôle

Ces critères s'expriment uniquement à partir du graphe de contrôle du programme testé. L'objectif de ce type de critère est de parcourir, à un coût raisonnable, le plus grand nombre possible de chemins de contrôle du programme. Les trois critères les plus utilisés dans cette catégorie sont les suivants :

- Le critère de *test des chemins* repose sur le parcours de tous les chemins possibles du programme. C'est le critère structurel le plus fort et il permettrait dans l'idéal de détecter les chemins infaisables, mais il est rarement applicable (en raison du nombre très grand, même infini le plus souvent, de chemins possibles). Des versions affaiblies de ce critère ont été proposées : le critère de *test des chemins structurés* consiste à parcourir tous les chemins du programme, de longueur inférieure ou égale à une valeur fixée. Nous reviendrons sur ce critère lorsque nous détaillerons le test des boucles.
- Le critère de *test des instructions* vise à exécuter au moins une fois chaque instruction (exécutable) du programme. C'est le critère structurel le plus faible et il est considéré comme la base minimale dans tout processus de test.
- Le critère de *test des branches* est intermédiaire. Il consiste à parcourir au moins une fois chaque branchement conditionnel du programme. Notons que le critère de test des branches inclut le critère de test des instructions : tout jeu de test satisfaisant le premier critère satisfait aussi le second [XMDA+94].

La Figure 2.1 montre différents jeux de test pouvant être associés à un programme selon les trois critères cités précédemment. Sur cette figure, on remarque qu'une donnée de test est suffisante pour exécuter toutes les instructions du programme, mais cela ne permet pas de tester les cas où les tests des conditionnelles s'évaluent à faux. Si l'on choisit le critère de test des branches, il faut deux données de test afin de couvrir tous les branchements conditionnels du programme, mais on ne teste toujours pas toutes les combinaisons que peuvent produire les valeurs des tests des conditionnelles. Le critère de test des chemins permet d'y remédier puisqu'avec quatre données de test, tous les chemins du programme sont exécutés.

Le critère de test des chemins est applicable au programme de la Figure 2.1 car son graphe de contrôle est simple et ne contient pas de boucles. Afin de pouvoir tester pratiquement et de la meilleure façon possible les programmes comprenant des boucles, des critères spécialisés ont été définis. Deux d'entre eux méritent d'être signalés : le critère de *test des chemins structurés* et le critère de *test bornes+corps des boucles.*

Le critère de *test des chemins structurés* consiste à fixer un entier k et à parcourir tous les chemins différents itérant au plus k fois chaque boucle.

Le critère de *test bornes+corps des boucles* consiste à faire du test des chemins classique sur les chemins ne contenant pas de boucles, et à appliquer un traitement

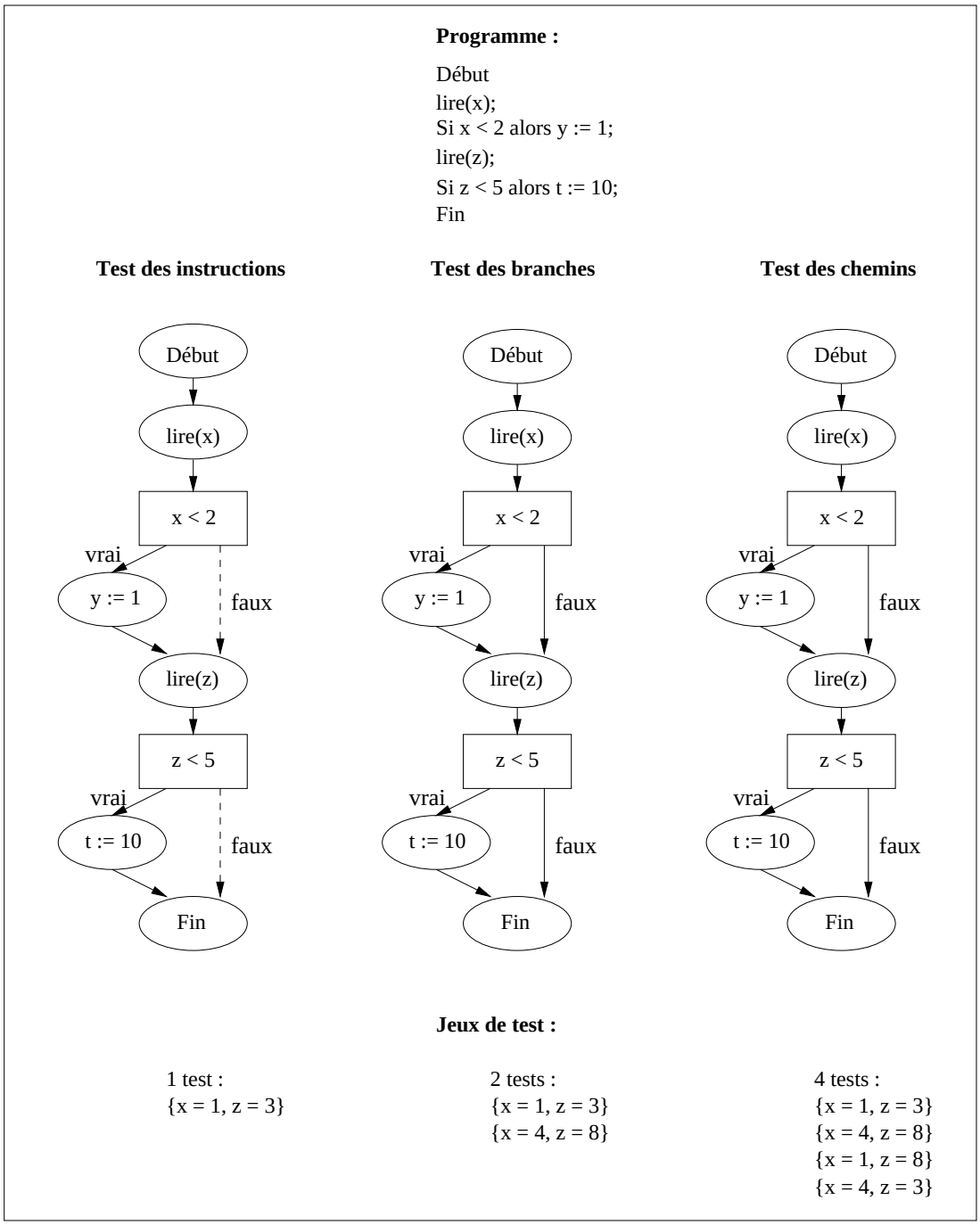


FIG. 2.1 – *Illustration des critères structurels.*

particulier aux chemins comprenant des boucles. Ce traitement regroupe deux sortes de test : le test de la valeur limite pour entrer dans la boucle et le test du corps de la boucle. Le test de la valeur limite conduit à choisir des données permettant de couvrir les différents chemins sans itération dans la boucle, alors que le test du corps nécessite des données permettant de suivre tous les chemins différents à travers la première itération de la boucle.

Lorsque le critère de test des chemins structurés est utilisé avec $k = 1$, la différence entre ce critère et celui de test bornes+corps des boucles est subtile. Ces deux critères ont en commun de permettre de suivre tous les chemins n'entrant pas dans la boucle, et tous les chemins entrant dans la boucle mais ne l'itérant pas. Leur différence réside dans le fait que le premier couvre aussi tous les chemins itérant *exactement* une fois la boucle, alors que le second couvre tous les chemins itérant *au moins* une fois la boucle. Ceci implique que tout jeu de test satisfaisant le critère de test des chemins structurés avec $k = 1$ satisfait aussi le critère de test bornes+corps des boucles (la réciproque étant fausse).

Nous illustrons le fonctionnement de ces deux derniers critères dans la Figure 2.2. Les données lues par le programme sont supposées être des entiers positifs. Le critère de test des chemins structurés est utilisé avec $k = 2$. Les deux premières données de test pour chaque critère correspondent à l'exécution des deux branches de la conditionnelle (cas où on entre dans la boucle mais sans faire d'itération). Les quatre suivantes correspondent aux quatre combinaisons possibles des branches de la conditionnelle lors de ses deux exécutions (cas d'une itération de la boucle). Parmi les sous-chemins correspondant à deux itérations de la boucle, certains sont infaisables : $(2-3-4-2-3-4-2-3-5-2)$, $(2-3-4-2-3-5-2-3-4-2)$, $(2-3-4-2-3-5-2-3-5-2)$, $(2-3-5-2-3-5-2-3-4-2)$ et $(2-3-5-2-3-5-2-3-5-2)$. Les trois données de test suivantes sont donc suffisantes pour tester l'ensemble des combinaisons possibles des branches de la conditionnelle lors de ses trois exécutions. Enfin, la dernière donnée de test permet de tester le cas où on n'entre pas dans la boucle.

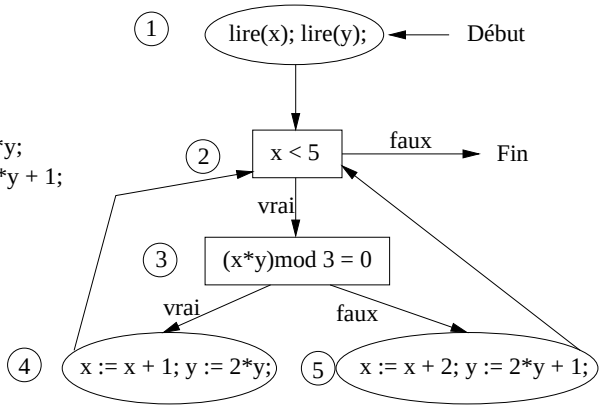
Notons que si le programme testé ne contient pas de boucle, le critère de test des chemins structurés et le critère de test bornes+corps des boucles sont équivalents au test des chemins.

Nous prenons un dernier exemple permettant d'illustrer l'ensemble des critères structurels de type *flot de contrôle* présentés dans cette partie : la procédure puissance (cf. Figure 2.3). On remarque l'existence d'un chemin infaisable (le chemin $(1-2-3-4-7-8-9)$).

- Pour satisfaire le critère de test des instructions, une donnée de test est suffisante : par exemple, $\{(x \in \mathbb{N}, y = -1)\}$.
- Pour satisfaire le critère de test des branches, deux données de test sont suffisantes : par exemple, $\{(x \in \mathbb{N}, y = 0), (x \in \mathbb{N}, y = -1)\}$.
- Pour satisfaire le critère de test des chemins, une infinité de données de test est nécessaire.

Programme :

```
Début
lire(x);
lire(y);
Tant que x < 5 faire
   Si (x*y)mod 3 = 0 alors x := x + 1; y := 2*y;
                     sinon x := x + 2; y := 2*y + 1;
Refaire;
Fin
```



Test des chemins structurés (k=2) :

Test des chemins n'itérant pas la boucle :
{x = 4, y = 3}
{x = 4, y = 2}

Test des chemins itérant 1 fois la boucle :
{x = 3, y = 7}
{x = 3, y = 6}
{x = 2, y = 4}
{x = 2, y = 5}

Test des chemins itérant 2 fois la boucle :
{x = 1, y = 1}
{x = 1, y = 2}
{x = 2, y = 0}

Test des chemins n'entrant pas dans la boucle :
{x = 6, y = 0}

→ 10 tests

Test bornes+corps des boucles :

Test de la valeur limite :
{x = 4, y = 3}
{x = 4, y = 2}

Test du corps :
{x = 2, y = 3}
{x = 3, y = 5}
{x = 2, y = 4}
{x = 2, y = 5}

Test des chemins n'entrant pas dans la boucle :
{x = 6, y = 0}

→ 7 tests

FIG. 2.2 – *Illustration des critères spécifiques aux boucles.*

- Pour satisfaire le critère de test des chemins structurés, avec $k = 2$, sept données de test sont suffisantes : par exemple, $\{(x \in \mathbb{N}, y = 1), (x \in \mathbb{N}, y = -1), (x \in \mathbb{N}, y = 2), (x \in \mathbb{N}, y = -2), (x \in \mathbb{N}, y = 3), (x \in \mathbb{N}, y = -3), (x \in \mathbb{N}, y = 0)\}$.
- Pour satisfaire le critère de test bornes+corps des boucles, cinq données de test sont suffisantes : par exemple, $\{(x \in \mathbb{N}, y = 1), (x \in \mathbb{N}, y = -1), (x \in \mathbb{N}, y = 2), (x \in \mathbb{N}, y = -2), (x \in \mathbb{N}, y = 0)\}$.

On définit une relation d'implication sur les critères de la manière suivante : un critère C_1 implique un critère C_2 (noté $C_1 \Rightarrow C_2$) si et seulement si pour tout programme, tout jeu de test satisfaisant C_1 satisfait aussi C_2. Pour résumer, on peut classer selon cette relation d'implication les critères que nous avons présentés jusqu'ici (Partie 2.1) :

Critère de test des chemins

$\Downarrow$

Critère de test des chemins structurés (avec $k \geq 1$)

$\Downarrow$

Critère de test bornes+corps des boucles

$\Downarrow$

Critère de test des branches

$\Downarrow$

Critère de test des instructions

On peut aussi associer aux critères une notion de coût comme, par exemple, le nombre minimal de données de test (c'est-à-dire le nombre de données du jeu de test le plus économique parmi ceux qui satisfont le critère) nécessaires dans le pire des cas (c'est-à-dire pour la structure de programme la plus défavorable). On obtient alors les résultats suivants pour les critères basés sur le flot de contrôle :

1. Le critère de test des chemins peut produire des jeux de test de taille infinie dans le pire des cas (programme contenant une boucle).
2. Le critère de test des chemins structurés et le critère de test bornes+corps des boucles produisent des jeux de test dont la taille est une fonction exponentielle du nombre d'instructions du programme dans le pire des cas.
3. Le critère de test des branches et le critère de test des instructions produisent des jeux de test dont la taille est une fonction linéaire du nombre d'instructions du programme.

Il va de soi que la relation d'implication définie plus haut entraîne un ordre sur le coût de critères. En d'autres termes, si C_1 implique C_2 alors C_1 est plus coûteux que C_2.

Il existe d'autres critères structurels intermédiaires que nous n'avons pas jugé utile de mentionner ici, car ils sont nettement moins utilisés. Néanmoins, le lecteur intéressé peut se référer à l'évaluation comparative de la qualité de différents critères structurels

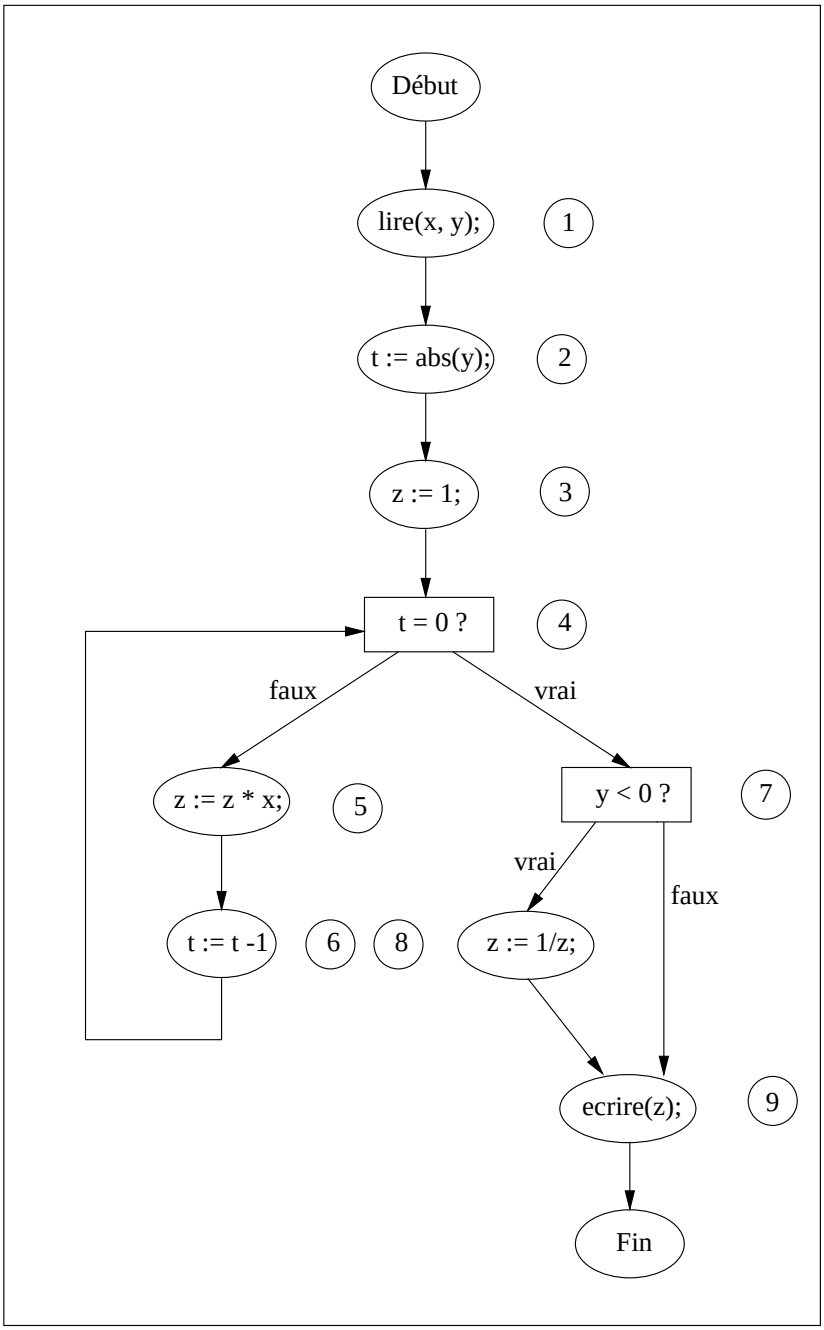


FIG. 2.3 – *Exemple récapitulatif (procédure puissance).*

basés sur le flot de contrôle présentée dans [NTA88]. Les critères basés sur le flot de contrôle permettent de détecter différentes erreurs telles que les erreurs de sélection de chemin (le chemin de contrôle effectivement traversé lors de l'exécution du programme n'est pas celui escompté) ou les erreurs de chemins inaccessibles (un chemin qui n'est jamais parcouru, quelle que soit l'entrée sur laquelle on exécute le programme). Ce sont en fait des erreurs sur le domaine (dans sens où le partitionnement du domaine d'entrée du programme en sous-domaines nécessitant le même calcul est mal réalisé). On peut améliorer la qualité du test en choisissant aussi de prendre en compte les erreurs dans le calcul. Pour cela, on peut utiliser le graphe de flot de données comme nous le montrons ci-après. En revanche, certaines erreurs comme les erreurs de chemins manquants sont en général hors de portée des jeux de test structurels. En effet, le chemin manquant étant absent du programme, il est indispensable de disposer d'une spécification afin de pouvoir produire au moins une donnée de test devant conduire au parcours de ce chemin.

2.2 Critères structurels basés sur le flot de données

On peut définir des critères structurels en exploitant le graphe de flot de données d'un programme. L'intérêt d'un critère basé sur les flots de données est d'engendrer des jeux de test qui permettent de détecter des erreurs dans le calcul effectué par le programme.

Un critère structurel de type *flot de données* [OW91] est basé sur la couverture (totale ou partielle) des associations définition/utilisation présentes dans le programme. On a une association définition/utilisation lorsqu'il existe une définition de variable, une utilisation de cette même variable et un chemin allant de la première à la seconde qui ne contient pas d'autre définition de cette même variable.

On peut citer deux critères principaux de type flot de données, le second étant bien plus faible que le premier :

- le test de *toutes* les associations définition/utilisation du programme (communément nommé *all-uses* dans la littérature) ;
- pour *chaque* définition présente dans le programme, le test d'*une* association définition/utilisation (*all-defs*).

Il existe aussi des variantes affaiblies du premier critère selon que l'utilisation de la variable se fait dans un prédicat (*p-utilisation*[1]) ou dans un calcul (*c-utilisation*) [RW85, NTA88] :

- critère de test de *toutes* les associations définition/c-utilisation et des associations définition/p-utilisation suffisantes pour que chaque définition du programme soit testée par au moins une association définition/utilisation quelconque (*all-c-uses/some-p-uses*) ;
- critère de test de *toutes* les associations définition/p-utilisation et des associations définition/c-utilisation suffisantes pour que chaque définition du programme

1. La couverture d'une p-utilisation implique de couvrir les deux branches de sortie du prédicat.

soit testée par au moins une association définition/utilisation quelconque (*all-p-uses/some-c-uses*) ;
- critère de test de *toutes* les associations définition/p-utilisation (*all-p-uses*).

Le dual du dernier critère (le test de *toutes* les associations définition/c-utilisation) n'est pas très intéressant car il ne permet ni la couverture des définitions du programme (ce qui est fait par les critères *all-c-uses/some-p-uses* et *all-p-uses/some-c-uses*), ni celle des branches (impliquée par le critère *all-p-uses*). Nous ne le considérons donc pas par la suite. Reprenons l'exemple de la procédure puissance, présentée dans la Figure 2.3 de la Partie 2.1. Voici la liste des définitions et utilisations apparaissant dans ce programme :

1. définition(x) et définition(y).
2. définition(t) et c-utilisation(y).
3. définition(z).
4. p-utilisation(t).
5. définition(z), c-utilisation(z) et c-utilisation(x).
6. définition(t) et c-utilisation(t).
7. p-utilisation(y).
8. définition(z) et c-utilisation(z).
9. c-utilisation(z).

Nous pouvons donc construire les associations définition/utilisation suivantes. Les associations définition/p-utilisation sont : $(1, 7 \rightarrow 8, y)$, $(1, 7 \rightarrow 9, y)$, $(2, 4 \rightarrow 5, t)$, $(2, 4 \rightarrow 7, t)$, $(6, 4 \rightarrow 5, t)$, $(6, 4 \rightarrow 7, t)$. Les associations définition/c-utilisation sont : $(1, 5, x)$, $(1, 2, y)$, $(2, 6, t)$, $(3, 5, z)$, $(3, 8, z)$, $(3, 9, z)$, $(5, 5, z)$, $(5, 8, z)$, $(5, 9, z)$, $(6, 6, t)$, $(8, 9, z)$. Notons que l'association définition/c-utilisation $(3, 8, z)$ est impossible car elle correspond à un chemin qui n'est pas exécutable. Nous dirons qu'elle n'est pas valide.

- Pour satisfaire les critères *all-uses*, *all-c-uses/some-p-uses*, *all-p-uses/some-c-uses* et *all-p-uses*, trois données de test sont suffisantes : par exemple, on peut choisir $\{(x \in \mathbb{N}, y = -1), (x \in \mathbb{N}, y = 0), (x \in \mathbb{N}, y = 2)\}$.
- Pour satisfaire le critère *all-defs*, une donnée de test est suffisante : par exemple, $\{(x \in \mathbb{N}, y = -1)\}$.

La difficulté majeure pour l'utilisation d'un critère de type flot de données est la détermination des associations définition/utilisation valides. En général, on ne peut pas déterminer précisément les associations définition/utilisation valides de façon statique. Dans ces conditions, l'utilisation d'un critère de type flot de données s'avère assez complexe. Ce problème a surtout été étudié dans le cas où le langage traité est rudimentaire (c'est-à-dire sans pointeurs ni variables structurées ni appels de procédures). Une théorie a toutefois été proposée pour un langage simple [RW85], puis étendue [RW85] pour traiter le langage Pascal, et enfin modifiée afin de prendre en compte les pointeurs [OW91].

L'idée de base de tous ces travaux est de définir différents types d'associations définition/utilisation selon que l'on est plus ou moins sûr de leur validité. On décompose les notions de définition en définition-possible et définition-sûre, et d'utilisation en utilisation-possible et utilisation-sûre. On utilise définition-possible (resp.

utilisation-possible) pour caractériser une définition (resp. une utilisation) d'une déréférence de pointeur dont l'ensemble d'alias contient plus d'une variable. Définition-sûre et utilisation-sûre sont utilisées dans les autres cas, c'est-à-dire quand on peut déterminer statiquement qu'une variable donnée a été définie ou utilisée. Mais pour avoir une association définition/utilisation, il faut aussi qu'il existe un chemin allant de la définition d'une variable à son utilisation et que ce chemin ne contienne pas d'autre définition (sûre ou possible) de cette variable: c'est un chemin sans-définition-sûre, ou aucune définition-sûre mais au moins une définition-possible: c'est un chemin sans-définition-possible. Le fait qu'un chemin soit sans définition est également indécidable en général. En combinant ces notions, on obtient quatre types d'associations définition/utilisation principaux:

- association définition/utilisation forte: définition-sûre et utilisation-sûre de la même variable, il existe au moins un chemin sans-définition (sûre et possible) et tout chemin sans-définition est sans-définition-sûre.
- association définition/utilisation solide: définition-sûre et utilisation-sûre de la même variable, il existe au moins un chemin sans-définition-sûre et au moins un chemin sans-définition-possible.
- association définition/utilisation faible: définition-sûre et utilisation-sûre de la même variable, il n'existe aucun chemin sans-définition-sûre.
- association définition/utilisation très faible: définition-possible et/ou utilisation-possible.

Le fait qu'une définition ou une utilisation soit jugée sûre (comme celui qu'un chemin soit sans définition sûre ou possible) résulte d'une analyse statique du code. Il est souhaitable de disposer d'une analyse précise afin d'obtenir le plus possible de définitions (ou utilisations) sûres. En prenant le cas extrême, une analyse imprécise, mais correcte, peut simplement indiquer que toutes les définitions et utilisations sont non sûres.

La Figure 2.4 présente un programme dont nous listons les différentes définitions et utilisations par instruction (en supposant une analyse exacte, c'est-à-dire la plus précise possible):

1. définition-sûre(x).
2. définition-sûre(y).
3. définition-sûre(p).
4. définition-sûre($*p$), définition-sûre(x) et utilisation-sûre(p).
5. définition-sûre(p).
6. définition-sûre(p).
7. définition-sûre($*p$), définition-possible(x), définition-possible(y), utilisation-sûre(p) et utilisation-sûre(y).
8. définition-sûre(z), utilisation-sûre(x) et utilisation-sûre(y).

À partir de ces définitions et utilisations, nous pouvons déterminer les associations définition/utilisation des quatre types définis précédemment qui sont présentes dans le

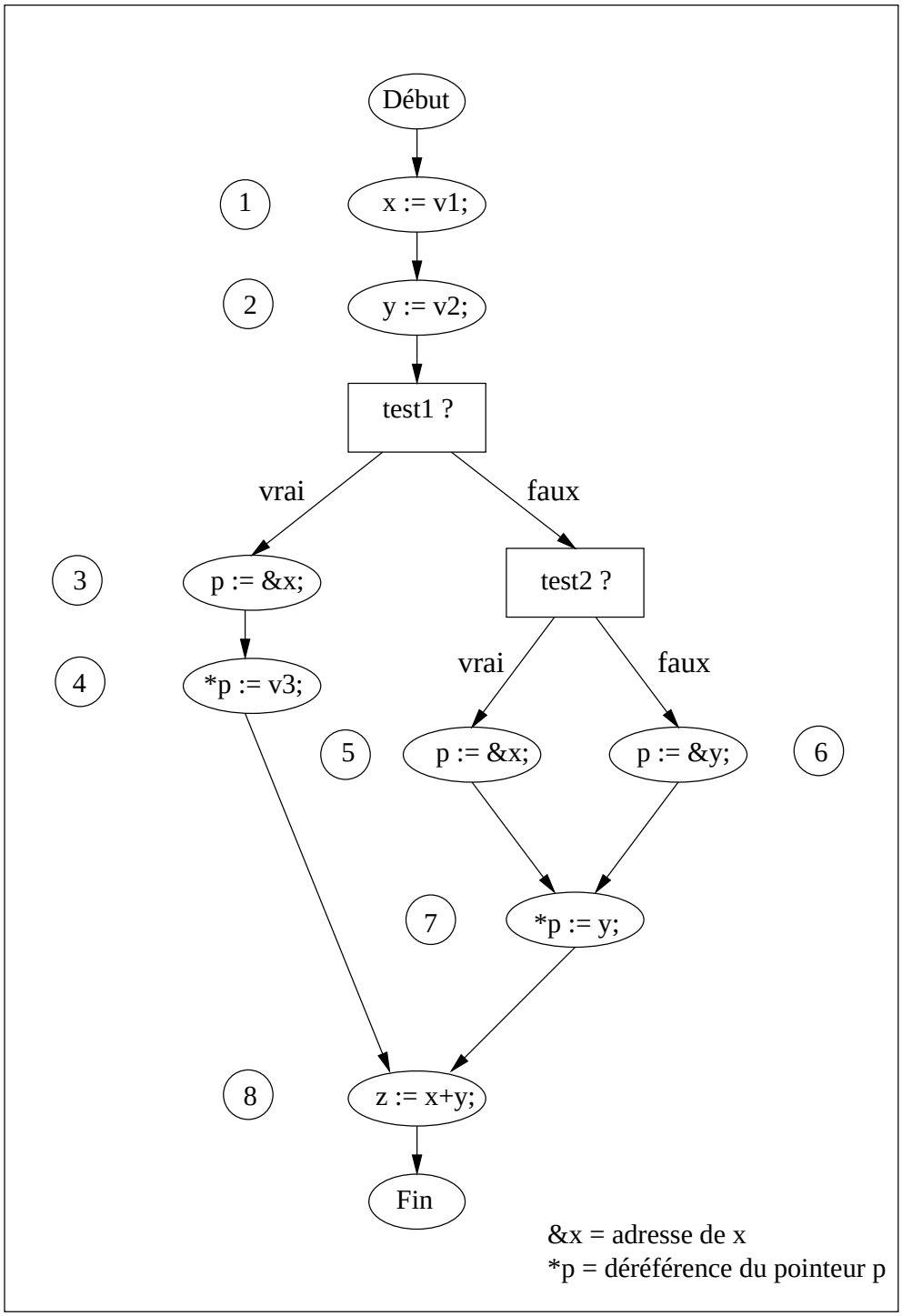


FIG. 2.4 – *Exemple pour le calcul des associations définition/utilisation.*

programme :

- associations définition/utilisation fortes : $(2, 7, y)$, $(3, 4, p)$, $(4, 8, x)$, $(5, 7, p)$ et $(6, 7, p)$.
- associations définition/utilisation solides : $(2, 8, y)$.
- associations définition/utilisation faibles : $(1, 8, x)$.
- associations définition/utilisation très faibles : $(7, 8, x)$ et $(7, 8, y)$.

Partant de ces nouvelles bases, les critères *all-uses* et *all-defs* cités plus haut ont été adaptés pour le traitement de langages avec pointeurs [OW91].

Critère toutes-définitions : Un jeu de test vérifie ce critère si *toutes* les définitions du programme sont exécutées lors du test. C'est-à-dire que l'on teste *une* association définition/utilisation quelconque pour *chaque* définition du programme.

Ce critère est en fait le critère *all-defs* qui n'a pas été invalidé par l'introduction des pointeurs. Dans le cas du programme de la Figure 2.4, il faut exécuter une association pour tester la définition de x dans l'instruction 1 (ici, $(1, 8, x)$), une association pour tester la définition de y dans l'instruction 2 (ici, $(2, 7, y)$ par exemple), une association pour tester la définition de p dans l'instruction 3 (ici, $(3, 4, p)$), une association pour tester la définition de x dans l'instruction 4 (ici, $(4, 8, x)$), une association pour tester la définition de p dans l'instruction 5 (ici, $(5, 7, p)$), une association pour tester la définition de p dans l'instruction 6 (ici, $(6, 7, p)$), et une association pour tester la définition de $*p$ dans l'instruction 7 (ici, $(7, 8, x)$ par exemple). Remarquons qu'on ne teste pas les définitions de $*p$ et z dans les instructions 4 et 8 car elles ne sont pas utilisées par la suite. Trois données de test suffisent donc. Une (telle que $test1 = vrai$) pour tester l'association $(3, 4, p)$, une autre (telle que $test1 = faux$ et $test2 = vrai$) pour tester l'association $(5, 7, p)$, et une dernière (telle que $test1 = faux$ et $test2 = faux$) pour tester l'association $(6, 7, p)$, les associations $(2, 7, y)$ et $(7, 8, x)$ étant aussi testées par ces deux derniers cas , tandis que l'association $(4, 8, x)$ l'est par le premier et l'association $(1, 8, x)$ par les trois.

Critère toutes-utilisations (*all-uses'*) : Ce critère est en fait l'union de quatre critères que l'on peut satisfaire indépendamment les uns des autres :

toutes-utilisations-fortes, *toutes-utilisations-solides*, *toutes-utilisations-faibles* et *toutes-utilisations-très faibles*.

L'un de ces critères est satisfait par un jeu de test si *toutes* les associations définition/utilisation correspondantes sont exécutées au cours du test.

On peut noter que ces critères pour le test des associations définition/utilisation sont d'importances différentes. D'une part, il y a les critères toutes-utilisations-fortes et toutes-utilisations-solides qu'il est indispensable de vérifier car ils permettent de tester des associations effectives. D'autre part, les critères toutes-utilisations-faibles et toutes-utilisations-très faibles dont la vérification est moins critique puisqu'ils ne permettent de tester que des associations possibles.

Si on reprend l'exemple de la Figure 2.4, les trois données de test trouvées précédemment suffisent encore pour satisfaire ce critère. En effet, l'association $(2, 8, y)$ est

testée par les trois cas, tandis que l'association $(7, 8, y)$ l'est par les deux derniers. Pour cet exemple, on constate que les critères toutes-définitions et toutes-utilisations sont équivalents.

Nous terminons la présentation des critères basés sur le flot de contrôle ou le flot de données en donnant leur classification dans la Figure 2.5. Cette classification est réalisée selon la relation d'implication définie précédemment.

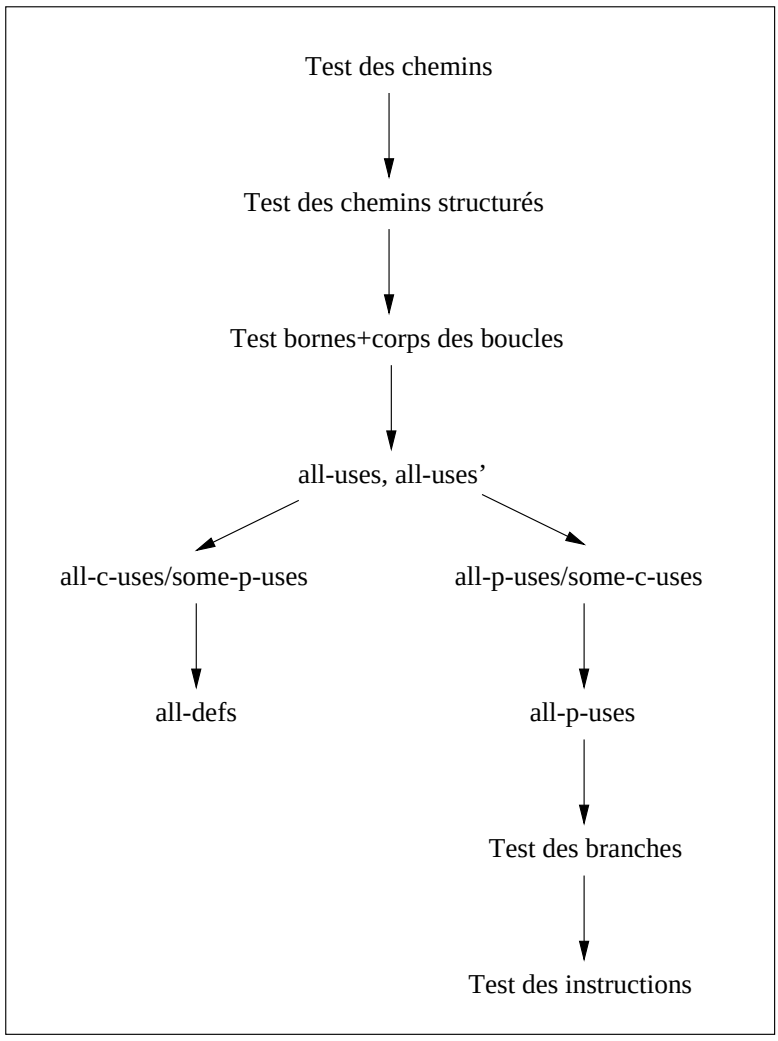


FIG. 2.5 – *Classification des critères structurels selon la relation d'implication définie page 25.*

Cette classification est partielle puisque certains critères ne sont pas comparables entre eux (*all-c-uses/some-p-uses* et *all-defs* avec *all-p-uses/some-c-uses*, *all-p-uses*, le critère de test des branches et le critère de test des instructions). Il est cependant intéressant de constater qu'il est possible de classer en une même hiérarchie des critères

de natures différentes puisque basés soit sur le flot de contrôle, soit sur le flot de données. Un exemple de cet entrelacement est donné par le fait que le critère de test bornes+corps des boucles implique les critères *all-uses* et *all-uses'*. Cela n'est pas surprenant dans la mesure où, considérant un programme sans boucle, le critère de test bornes+corps des boucles entraîne la couverture de tous les chemins, et donc celle de toutes les associations définition/utilisation ; et considérant un programme avec boucles, ce critère conduit aussi à couvrir :

- d'une part tous les chemins traversant exactement une fois le corps de la boucle, ce qui permet de couvrir les associations définition/utilisation constituées d'une définition située dans le corps de la boucle et d'une utilisation postérieure dans la même étape de la boucle ou hors de la boucle, ainsi que celles constituées d'une définition antérieure à la boucle et d'une utilisation dans la boucle ;
- d'autre part tous les chemins exécutant au moins deux fois le corps de la boucle, ce qui permet de couvrir les associations définition/utilisation constituées d'une définition située dans le corps de la boucle et d'une utilisation lors de l'étape suivante de la boucle.

Nous en avons un autre exemple avec l'implication du critère de test des branches par *all-p-uses*. Ce dernier consiste a couvrir toutes les associations définition/p-utilisation. Or, chaque branchement conditionnel correspond à au moins une p-utilisation (dans le cas où aucune des deux branches n'est impossible). La couverture des associations définition/p-utilisation entraîne donc le parcours de toutes les branches (puisque couvrir une p-utilisation consiste à couvrir deux chemins correspondant aux deux valeurs possibles du prédicat faisant la p-utilisation).

Les critères cités ici sont les plus représentatifs et les plus utilisés dans leur catégorie, néanmoins, pour une classification plus complète de ce type de critères, on peut se référer aux travaux présentés dans [NTA88, CPRZ89].

Un critère de type flot de données utilisé en complément d'un critère basé sur le flot de contrôle permet de couvrir un assez large spectre d'erreurs. Mais l'une des principales sources d'erreurs dans un programme est le traitement des valeurs aux limites (dans les conditionnelles, les boucles) et ce type d'erreur est peu, voire pas du tout, détecté par les critères structurels basés sur le flot de contrôle ou le flot de données. On peut penser que, de par leur nature, ce sont les critères fonctionnels qui sont les plus indiqués pour le test aux limites. Il existe cependant un critère structurel qui peut englober le test aux limites, c'est celui *d'élimination des mutants* que nous détaillons à présent.

2.3 Critères structurels basés sur les fautes

Nous avons déjà mentionné que les critères structurels ne sont pas tous basés sur le flot de contrôle ou le flot de données. C'est le cas des critères basés sur les fautes, ces fautes étant définies à partir du programme et plus particulièrement, de sa syntaxe. Le but d'un tel critère est de permettre de couvrir chaque faute répertoriée. Le critère le plus connu et utilisé dans cette catégorie est le critère d'élimination des mutants [DO91].

La motivation du critère d'élimination des mutants est *l'hypothèse du programmeur compétent*. C'est-à-dire que l'on suppose que la structure du programme testé est globalement bonne mais qu'il peut contenir des fautes locales. Les fautes considérées sont donc des légères variantes syntaxiques du programme correct, des fautes simples. On peut introduire une distinction entre deux grandes classes de fautes : les fautes simples (modélisées par des opérateurs de mutation) et les fautes complexes (code manquant, placement incorrect, algorithme incorrect...). Les tenants du critère de mutation avancent l'argument que les fautes complexes sont généralement couplées à des fautes simples [DLS78]. Ainsi, les fautes complexes sont aussi détectées par un jeu de test destiné à dévoiler les fautes simples. Bien tester un programme revient alors à trouver les fautes simples qu'il contient. Pour chaque langage, on définit un ensemble d'opérateurs de mutation correspondant à l'ensemble des modifications syntaxiques considérées (remplacement d'un opérateur unaire par un autre opérateur unaire, d'un opérateur binaire par un autre opérateur binaire, d'un identificateur de variable par un autre identificateur, modification d'une valeur de constante...). En appliquant ces opérateurs de mutation au programme testé, on obtient des programmes (appelés mutants) qui ne diffèrent du premier que par une seule modification syntaxique. C'est la génération des mutants. Un jeu de test vérifiant le critère d'élimination des mutants doit permettre de distinguer tout mutant du programme de départ. On dit que le jeu de test élimine (ou tue) les mutants.

Dans la Figure 2.6, nous montrons un exemple de programme et le mutant qui lui est associé à partir de l'opérateur de mutation «l'opérateur binaire - est remplacé par l'opérateur binaire +». Nous expliquons ensuite comment trouver une donnée de test permettant d'éliminer ce mutant. Pour obtenir un jeu de test adéquat au critère d'élimination des mutants, on doit répéter ce processus pour chaque opérateur de mutation.

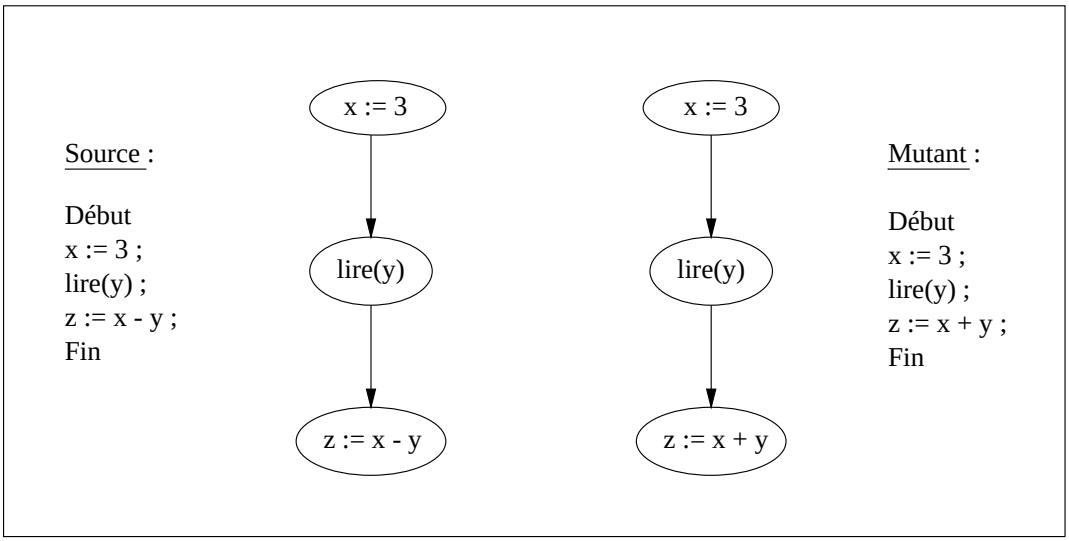


FIG. 2.6 – *Illustration du critère d'élimination des mutants.*

Pour que la donnée de test élimine le mutant, il faut qu'elle permette de distinguer

le mutant du code source et donc qu'elle vérifie la contrainte $(x = 3) \wedge ((x-y) \neq (x+y))$ c'est-à-dire $y \neq 0$. Cette contrainte est la conjonction des contraintes rencontrées sur le chemin menant à l'instruction mutante ainsi qu'une contrainte (appelée contrainte de nécessité) qui permet de distinguer l'instruction mutante de sa version originale. On peut, par exemple, choisir $\{y = 1\}$ comme donnée de test.

À présent, montrons comment cette donnée de test détecte effectivement une faute possible sur l'opérateur binaire. Admettons que le résultat du test du programme avec l'entrée $\{y = 1\}$ soit $z = 2$. Deux cas sont envisageables :

- si le programme source est correct (c'est donc une soustraction), l'oracle donne aussi $z = 2$ comme résultat : le test ne détecte pas d'erreur et c'est normal.
- si le programme source est incorrect (c'est donc une addition selon l'opérateur de mutation choisi), l'oracle donne $z = 4$ comme résultat et l'erreur est détectée : le test est réussi.

Les opérateurs de mutation sont choisis pour détecter de nombreux types d'erreurs. C'est ainsi que le critère d'élimination des mutants peut impliquer d'autres critères structurels tels que la couverture des branches (grâce aux mutations sur les prédicats) et par conséquent la couverture des instructions [DO91]. Il permet également de tester le passage par des valeurs limites puisque les opérateurs de mutation prennent en compte les cas particuliers des opérateurs prédéfinis du langage (par exemple, l'opérateur $\geq$ mute en $=$, $\neq$, $>$ et $\leq$).

On peut noter que la qualité des jeux de test dépend entièrement du choix des opérateurs de mutation. Ces opérateurs résultent d'un long travail constitué de raffinements successifs et d'évaluations. Jusqu'ici, des ensembles d'opérateurs de mutation ont été choisis pour les langages Fortran [KO91], C [ADH+89] et Ada [OPV96].

Notons aussi que ce critère est à la base de l'analyse de mutation [DLS78, HAM77], technique très souvent utilisée pour juger de la qualité d'un jeu de test [BUD81], et bénéficiant d'une reconnaissance importante dans la littérature (notamment pour évaluer la pertinence de divers critères de test et les comparer). L'analyse de mutation mesure un score de mutation, qui correspond à l'adéquation d'un jeu de test par rapport au critère d'élimination des mutants [DKM+89]. La mesure d'adéquation consiste à générer des mutants à partir du programme sous test, puis à exécuter tous ces mutants sur le jeu de test dont on veut évaluer l'adéquation. Tous les mutants pour lesquels le résultat de l'exécution est différent de celui donné par l'exécution du jeu de test sur le programme d'origine sont déclarés éliminés (ou tués). Normalement, pour qu'un jeu de test soit adéquat, il faudrait que tous les mutants soient éliminés puisque chacun est censé représenter un codage incorrect. L'inconvénient est que certains mutants, bien que syntaxiquement différents du programme sous test, peuvent lui être sémantiquement équivalents. Dans ce cas, ils ne peuvent pas être éliminés. On doit donc enlever manuellement les mutants équivalents au programme testé des mutants pouvant rester à l'issue de l'analyse de mutation. Finalement, les mutants restant permettent de révéler l'inaptitude du jeu de test à détecter certaines fautes. Le score de mutation est ainsi défini comme le rapport entre le nombre de mutants éliminés et le nombre de mutants

générés. Cette démarche est séduisante mais présente deux inconvénients majeurs :

- le nombre de mutants générés est considérable[2], ce qui fait que leur exécution peut devenir vraiment coûteuse ;
- l'équivalence d'un mutant et du programme d'origine n'est pas décidable.

Nous constatons donc que ce problème d'indécidabilité se retrouve sous des formes variées dans tous les critères structurels (chemins impossibles, définitions/utilisations non sûres, équivalence de mutants...). Nous reviendrons sur cette observation importante dans le Chapitre 4 et dans la présentation de la démarche adoptée dans cette thèse (Chapitre 1 de la Partie II).

Nous avons exposé un certain nombre de critères de test parmi les plus utilisés. Il existe en fait un très grand nombre de critères possibles. Il convient donc de caractériser les qualités attendues d'un bon critère de test afin d'être capable d'opérer un choix dans cette palette de possibilités. Nous présentons deux manières d'aborder ce problème dans la Partie 2.4.

2.4 Évaluation d'un critère de test

Deux types d'évaluations des qualités des critères de test ont été proposées dans la littérature : les premières reposent sur des caractérisations de nature théorique et les secondes font appel à des résultats expérimentaux. Nous les présentons succintement dans cet ordre dans les Parties 2.4.1 et 2.4.2.

2.4.1 Évaluation théorique

La principale qualité d'un bon critère de test est de permettre d'engendrer des jeux de test robustes [GG75]. Un jeu de test est robuste si le succès de son exécution (c'est-à-dire si les résultats obtenus après l'exécution du test sont conformes à ceux attendus) implique que le programme testé ne contient pas de fautes. Les propriétés de fiabilité et de validité ont été introduites pour raffiner cette notion de robustesse [GG75]. On dit qu'un critère C est fiable si pour tous jeux de test T_1, T_2 dérivés du programme P satisfaisant C, le test de P par T_1 produit un résultat cohérent avec celui du test de P par T_2. Cette cohérence est exprimée par une équivalence entre le succès de T_1 et le succès de T_2. On dit que C vérifie la propriété de validité si pour toute faute dans P, il existe un jeu de test T satisfaisant C capable de la détecter.

On peut remarquer que si un critère C est fiable et valide alors, pour tout programme P, tout jeu de test T satisfaisant C est capable de détecter toute faute dans P ; T est donc robuste. En effet, procédons par l'absurde et imaginons qu'il existe un programme P contenant une faute, et qu'il existe un jeu de test T satisfaisant C ne la détectant pas. Comme C est fiable, aucun jeu de test satisfaisant C ne la détecte. Cela contredit donc le fait que C est valide.

2. Il semble que le nombre de mutants soit quadratique par rapport au nombre de lignes de code, au nombre de branches, au nombre de variables et de références de variables [ORZ93].

La propriété de robustesse est très forte puisque sa vérification revient à assurer la correction du programme, qui est indécidable en général. Il est donc nécessaire de l'affaiblir afin de pouvoir l'utiliser en pratique. La notion d'hypothèse de test a été introduite pour caractériser cet affaiblissement [GAU95]. Par exemple, un critère de test basé sur le partitionnement du domaine d'entrée en sous-domaines correspond à une hypothèse d'uniformité [GAU95], qui suppose que toutes les valeurs d'un même sous-domaine entraînent un même comportement du programme testé. Avec cette hypothèse, il suffit donc de choisir une donnée par sous-domaine pour tester entièrement le domaine d'entrée du programme. Lorsque le test est réalisé en exécutant le programme sur des structures de données de petite taille (listes de longueur inférieure à deux par exemple), l'hypothèse correspondante est une hypothèse de régularité qui suppose que, si le programme se comporte correctement sur toutes les valeurs de taille inférieure au seuil choisi, alors il se comporte de la même manière pour toutes les valeurs du domaine d'entrée.

Il existe une alternative à la démarche qui consiste à définir un concept de robustesse théorique, puis à l'affaiblir par des hypothèses de test pour son utilisation pratique. Elle repose sur la notion d'adéquation relative d'un jeu de test pour un programme par rapport à une classe donnée de programmes [DLS78, DO91]. Plus précisémment, si M est une classe de programmes et S une spécification sur un domaine D, on définit l'adéquation relative d'un jeu de test T pour S par rapport à M de la manière suivante :

$T \subset D$ est adéquat pour S relativement à M si pour tout programme $Q \in M$,
$\exists d \in D$ tel que $Q(d) \neq S(d) \Rightarrow \exists t \in T$ tel que $Q(t) \neq S(t)$.

La classe de programmes M peut être définie de différentes manières selon le type de fautes que l'on souhaite identifier. La méthode que nous présentons dans la seconde partie de ce document, par exemple, entre dans le cadre de l'adéquation relative en définissant M comme l'ensemble des instances d'un schéma de programmes donné. En pratique, on a surtout utilisé l'adéquation aux mutants [DLS78, DO91], la classe de programmes étant définie par mutation à partir du programme à tester (cf. Partie 2.3). Cette notion permet d'atteindre une robustesse limitée (par l'ensemble fini des mutants M de la définition), mais présente l'avantage d'être directement utilisable en pratique. Si l'hypothèse du programmeur compétent est vérifiée (cf. Partie 2.3), l'adéquation aux mutants est équivalente à la robustesse définie précédemment, et tout jeu de test généré selon le critère d'élimination des mutants est robuste.

Ainsi, un jeu de test et le critère qui a servi à le produire sont étroitement liés. Ce lien doit être explicité quand les jeux de test sont générés automatiquement, mais il reste en général implicite dans la pratique industrielle du test, ce qui interdit toute évaluation correcte du sens et de l'intérêt du test effectué. Afin de combler cette lacune, la notion de *contexte de test* [GAU95, BGM91] a été introduite. Un contexte de test est une paire (H, T) où H est un ensemble d'hypothèses sur le programme et T un jeu de test. La nature de H correspond en fait au choix du critère par lequel est généré T, qui est robuste sous réserve que H soit vérifié.

Nous concluons cette partie en remarquant que l'association d'hypothèses de test au

critère est indispensable pour déterminer de façon précise les conditions sous lesquelles le test est significatif et par conséquent ses limites [GAU95, BGM91]. Cependant, si cette précision est satisfaisante d'un point de vue théorique, ce n'est pas forcément exploitable en pratique. En effet, les hypothèses que l'on est amené à définir sont de nature sémantique, et il est généralement impossible de déterminer formellement si elles sont vérifiées pour un programme donné. Cela nous conduit à aborder le problème d'évaluation des critères sous un autre angle : en se basant sur des résultats expérimentaux.

2.4.2 Évaluation pratique

De nombreux travaux ont tenté de proposer des comparaisons entre les critères de test, en se basant sur des expérimentations ou des simulations. Cependant, il ne se dégage pas de réel consensus sur la question et il est difficile d'avoir une vision d'ensemble des critères, car les différentes études ne traitent le plus souvent que deux ou trois critères à la fois. En fait, l'évaluation pratique des critères de test est largement reconnue comme difficile, et des réserves ont notamment été émises quant à la validité d'expérimentations statistiques [HAM89]. Illustration de la difficulté de cette évaluation pratique, des travaux [BS87] ont conclu que le pouvoir de détection d'erreurs d'un critère ne dépend pas seulement du critère lui-même mais aussi du testeur (ou du choix d'un jeu de test parmi tous ceux qui satisfont le critère), des programmes choisis pour l'évaluation, et du type de fautes considéré.

Il faut également remarquer que les critères théoriques (tels ceux présentés dans les Parties 2.1, 2.2 et 2.3) ne sont pas utilisables directement en pratique car les chemins (branches, instructions...) ne sont pas tous exécutables. En pratique, *couvrir toutes les instructions* se transforme donc implicitement en *couvrir toutes les instructions exécutables*. Ce faisant, la relation d'implication définie pour les critères théoriques dans la Partie 2.1 n'est pas forcément conservée [RW85, FW88, NTA88]. De plus, il est prouvé [FW93b] que le fait qu'un critère C_1 implique un critère C_2 n'entraîne pas toujours un meilleur pouvoir de détection d'erreurs de C_1. Par exemple, le critère *all-uses* (test de toutes les associations définition/utilisation) implique le test des branches mais a parfois un moins bon taux de détection d'erreurs que ce dernier [FW93a].

En fait, les évaluations pratiques sont trop imprécises pour permettre de distinguer entre des méthodes de test mettant en oeuvre des critères trop proches [HAM89]. C'est le cas, par exemple, de l'ensemble des méthodes de test basées sur des critères structurels. Les travaux existant comparent plutôt des méthodes de test statique (sans exécution du programme) avec des méthodes de test dynamique [BS87], ou alors différents types de méthodes dynamiques (structurelles, fonctionnelles ou statistiques). Un certain nombre de travaux concerne ainsi la comparaison de critères de test statistique et de test des chemins [DN84, HT90, TFWC91]. Par exemple, le résultat principal de [DN84] est que le pouvoir de détection d'erreurs de 100 données de test simulant le test aléatoire est supérieur à celui de 50 données de test simulant le test des chemins. Bien que contre-intuitif, ce résultat a été conforté par la suite [HT90]. Il faut cependant le modérer par le fait que les résultats du test aléatoire sont très irréguliers (selon la

distribution de probabilité du domaine d'entrée du programme).

Chapitre 3

Techniques structurelles de génération de jeux de test

Dans le chapitre précédent, nous avons présenté et comparé différents critères structurels. Nous abordons maintenant les manières d'utiliser effectivement ces critères. Comme nous l'avons souligné plus avant, on peut distinguer deux options principales :

- Une génération de jeux de test par des moyens quelconques suivie d'une évaluation a posteriori de son adéquation à un critère donné.
- Une génération de jeux de test tenant compte d'un critère donné (et produisant donc des jeux de test adéquats pour ce critère).

La première option est plus facile à implanter, et elle a ainsi conduit à un certain nombre d'outils industriels. Pour l'évaluation des critères structurels basés sur le flot de contrôle ou le flot de données, il existe des outils de couverture des graphes de contrôle et de flot de données (LOGISCOPE de la société VERILOG, ASSET [FWW85], TACTIC [OW91]...). Par ailleurs, il existe aussi des outils d'analyse de mutation (MOTHRA [DKM+89], Teletype [DM95], PiSCES [VMM91]...). Les outils de cette catégorie offrent une aide non négligeable au testeur mais n'allègent pas la tâche, souvent fastidieuse, de production des données de test.

La seconde option est plus ambitieuse puisqu'elle implique un processus de génération (automatique dans le cas idéal) de données de test. Nous détaillons dans le reste de ce chapitre les différentes techniques proposées pour cette génération. La Partie 3.1 est consacrée à la génération de jeux de test à partir du simple texte source d'un programme, et la Partie 3.2 montre comment il est possible d'utiliser, conjointement au programme, une spécification.

3.1 Génération de jeux de test à partir du code des programmes

En général, les techniques reposant sur des critères structurels procèdent selon un schéma commun qui est le suivant :

1. La construction du graphe représentant le programme (ce graphe intègre le flot de contrôle et, si le critère le demande, le flot de données du programme).
2. La sélection des chemins vérifiant le critère.
3. La génération des données permettant de parcourir ces chemins (c'est-à-dire satisfaisant les contraintes associées).

Ces techniques peuvent être statiques ou dynamiques. Le principal travail sur la génération dynamique de jeux de test [KOR90] repose sur une résolution dynamique des contraintes. Cette résolution consiste à produire d'abord aléatoirement des données de test, à les exécuter sur le programme, puis à les modifier après chaque exécution de façon à ce que, in fine, elles vérifient les contraintes imposées. Cette méthode a un coût élevé car elle nécessite un nombre important d'exécutions du programme. En cela, les méthodes statiques sont plus économiques. Nous présentons ici trois types de méthodes statiques pour la génération de jeux de test reposant sur les différents critères de test structurels introduits dans le Chapitre 2. Nous étudions d'abord l'utilisation de l'analyse statique pour des critères visant à couvrir uniquement des propriétés du graphe de contrôle, puis nous considérons des critères prenant en compte le graphe de flot de données. Nous terminons par une méthode de test qui exploite le critère structurel d'élimination des mutants.

3.1.1 Test basé sur le flot de contrôle

Les techniques de génération de jeux de test qui reposent sur ce type de critère exploitent le graphe de contrôle afin de déterminer les contraintes à vérifier pour pouvoir parcourir un chemin donné, une branche donnée ou atteindre une instruction donnée. Différentes méthodes sont ensuite utilisées pour résoudre les contraintes. Ce processus peut être réalisé de manière statique [VA98] à l'aide de résolveurs de contraintes (tel que ILOG Solver de la société ILOG), ou bien en partie de manière dynamique afin de retarder ou d'éviter leur utilisation [KOR90]. Notons toutefois que les contraintes engendrées ne peuvent pas toujours être résolues automatiquement, ce qui peut conduire à des interactions entre le résolveur de contraintes et l'utilisateur.

3.1.2 Test basé sur le flot de données

L'utilisation d'un critère basé sur le flot de données conduit à engendrer des jeux de test permettant d'exécuter certaines des associations définition/utilisation présentes dans le programme testé. Nous rappelons que l'on a une association définition/utilisation lorsqu'il existe une définition de variable, une utilisation de cette même variable et un chemin allant de la première à la seconde sans exécuter une autre définition de cette même variable (cf. Partie 2.2).

Le système TACTIC [OW91] permet d'extraire les associations définition/utilisation d'un programme et d'évaluer la qualité d'un jeu de test par rapport à un critère basé sur le flot de données. TACTIC est initialement conçu pour mesurer l'adéquation d'un jeu de test aux critères de type flot de données et n'est pas destiné à faire de la génération de jeux de test. Néanmoins, il propose une représentation graphique des différentes associations définition/utilisation du programme de façon à ce que l'utilisateur puisse voir celles que son jeu de test n'exécute pas, et ainsi ajouter une nouvelle donnée de test pour compléter le jeu de test initial. Il pourrait donc être complété par un processus de résolution de contraintes comme il a été évoqué plus haut, pour produire des données de test de manière assistée.

3.1.3 Test basé sur les mutants

L'analyse de mutation est le plus souvent utilisée pour mesurer la couverture d'un jeu de test a posteriori, la génération de données de test en elle-même devant être effectuée par d'autres moyens. La méthode décrite ci-après est sans doute la première à permettre d'engendrer des données de test par l'analyse de mutation de façon automatique [DO91].

Le principe est d'engendrer un jeu de test permettant d'éliminer les mutants dérivés du programme original : c'est-à-dire que son exécution sur chaque mutant non équivalent au programme testé doit aboutir à un échec. En effet, quand des mutants ne sont pas éliminés par l'exécution d'un jeu de test, ils permettent de révéler l'inaptitude du jeu de test à détecter une erreur possible, ou alors la présence d'une faute dans le programme. Pour utiliser les mutants dans la génération de jeux de test en elle-même, on exprime par des contraintes algébriques les conditions sous lesquelles le comportement d'un mutant est différent de celui du programme correct (conditions d'élimination du mutant). On peut alors distinguer deux types de conditions : les conditions de nécessité et les conditions de suffisance.

- Une condition de nécessité implique que l'état du mutant de P juste après une exécution de l'instruction mutante doit être différent de l'état de P au même point.
- Une condition de suffisance implique que la sortie obtenue lors de l'exécution du mutant de P doit être incorrecte, c'est-à-dire que l'état final du mutant doit différer de celui de P.

Il est relativement facile de trouver une condition nécessaire associée à chaque mutant. En fait, un mutant de P se différencie de P par un changement syntaxique dû à un opérateur de mutation qui a été appliqué à P. À partir de cet opérateur, il est donc possible de dériver une condition de nécessité.

En revanche, trouver une condition suffisante est beaucoup plus difficile, et même impossible en général. En effet, cela signifierait que l'on connaît par avance le chemin que va suivre le programme. La plupart des travaux qui se sont heurtés à ce problème ont opté pour diverses approximations afin d'aboutir à des résultats pratiques. Par exemple, certains se contentent de conditions de nécessité, considérant que la plupart des données de test vérifiant ces dernières vérifient aussi les conditions de suffisance [DO91]. Cette

hypothèse semble d'ailleurs vérifiée dans la majorité des cas rencontrés en pratique. Néanmoins, lorsque la mutation porte sur un prédicat, on étend la condition de nécessité par la condition suivante : la valeur prise par le prédicat mutant doit être différente de celle du prédicat d'origine.

À un langage de programmation donné est associé un ensemble d'opérateurs de mutation à partir desquels on peut produire automatiquement des contraintes de nécessité. Il reste alors à engendrer les données de test qui satisfont ces contraintes. La satisfaction des contraintes est un problème difficile. Pour cela, des algorithmes et heuristiques (réduction du domaine) ont été développés [DO91], mais ne se révèlent efficaces que sur des contraintes simples (linéaires par exemple). À l'usage, il s'avère que cette méthode produit des jeux de test très pertinents [DO91] puisqu'ils sont adéquats au critère d'élimination des mutants par construction, et que ce critère couvre de nombreux types d'erreurs (cf. Partie 2.3). Il existe un outil (Godzilla [DO91]) qui implante cette technique pour les langages Fortran 77 et C.

3.2 Utilisation conjointe des spécifications et du code pour la génération de jeux de test

Nous avons décrit jusqu'ici des méthodes de génération de jeux de test basées sur l'analyse statique du programme à tester. Nous abordons maintenant des méthodes qui nécessitent une spécification formelle du programme, et qui utilisent conjointement les techniques de test et de preuve de programme. La combinaison de ces deux techniques complémentaires apparaît comme une démarche prometteuse pour évaluer le comportement d'un programme. Les travaux explorant cette voie se distinguent les uns des autres de deux manières. Premièrement, par le type des spécifications prises en entrée : certains considèrent la spécification comme un ensemble d'assertions (chacune caractérisant l'état du programme en un point donné [CAR81]), ce qui permet une approche locale du test ; d'autres choisissent d'appréhender la spécification dans sa totalité, ce qui conduit à tester le programme globalement [RC85]. Deuxièmement, ces travaux se différencient par la façon dont ils gèrent la génération des données de test. Dans un cas, l'évaluation symbolique de conditions issues de la spécification permet de dériver des données de test, sans avoir recours à un critère de test particulier [CAR81] ; dans l'autre cas, des critères de test prédéfinis sont utilisés [RC85].

Dans ce qui suit, nous détaillons deux méthodes combinant preuve et test de programme, illustrant chacune l'un des choix précédents.

3.2.1 Le test formel

Le test formel [CAR81] utilise des techniques issues de la vérification formelle de programmes pour sélectionner des jeux de test. Un système de test formel procède en quatre phases :

– l'insertion de spécifications formelles dans le code (assertions d'invariants),

- la génération des conditions de vérification associées à ces assertions par un système de vérification,
- leur simplification,
- le test des conditions de vérification.

On peut noter que le test formel et la vérification de programmes ne diffèrent que par la quatrième étape : on évalue chaque condition de vérification sur un jeu de test au lieu de la prouver. Ainsi, la qualité du test formel dépend en grande partie de celle des données de test utilisées dans cette dernière étape. Il est donc nécessaire d'évaluer l'adéquation des données de test générées pour une certaine condition de vérification, par exemple en ayant recours à l'analyse de mutation.

La génération des données de test pour une condition de vérification se fait en évaluant symboliquement cette condition tout en retardant le plus possible l'association de valeurs aux différentes variables. Lorsqu'un prédicat rend indispensable la liaison d'une variable, on choisit la valeur à lui associer dans un ensemble restreint E qui a été déterminé à partir d'heuristiques dépendant de ce prédicat (on utilise de nouveau des techniques de preuve automatique de programmes). Par exemple, si la variable appartient à un type inductif, l'ensemble E contient les cas de base et du premier niveau de sa définition (c'est-à-dire, pour le type *list*, *nil* et une liste quelconque de longueur 1). Quand, à l'issue de ce processus, on obtient une donnée de test, on procède par retour-arrière afin que chacune des valeurs présentes dans l'ensemble E intervienne dans au moins une donnée de test.

Prenons un exemple pour illustrer le fonctionnement de la génération des données de test. Soit la condition de vérification $P_1 \wedge (\text{ si } P_2 \text{ alors } P_3 \text{ sinon } P_4)$ avec $P_1 \equiv (x < y)$, $P_2 \equiv (pair?(x) = vrai)$, $P_3 \equiv (y < 10)$ et $P_4 \equiv (y = 20)$. Lors de l'évaluation symbolique de cette condition, le prédicat P_1 ne nécessite pas d'instancier les variables x et y. Par contre, le prédicat P_2 oblige à choisir une valeur pour la variable x (afin de déterminer quelle branche de la conditionnelle est parcourue). Imaginons une heuristique pour le prédicat P_2 qui détermine l'ensemble de valeurs pour x : $E_2 = \{0, 1, 2\}$. Cet ensemble permet de tester la valeur particulière 0, une valeur impaire et une valeur paire. On prend de manière quelconque une valeur dans cet ensemble : $x = 2$ par exemple. On termine l'évaluation de la condition par le prédicat P_3 qui ne nécessite pas d'instanciation immédiate. À ce stade, la condition peut être réécrite en $(x < y) \wedge (x = 2) \wedge (y < 10)$. Il suffit alors d'instancier pour obtenir une donnée de test : par exemple $(x = 2, y = 3)$. À présent, il faut faire un retour-arrière de façon à utiliser une autre valeur de l'ensemble E_2 (intuitivement, cela correspond à tester toutes les valeurs jugées intéressantes par l'heuristique associée à P_2) : par exemple $x = 1$. L'évaluation de la condition se termine alors par le prédicat P_4 et se réécrit en $(x < y) \wedge (x = 1) \wedge (y = 20)$, ce qui conduit à la donnée de test $(x = 1, y = 20)$. Un dernier retour-arrière est nécessaire pour couvrir toutes les valeurs de l'ensemble E_2, et on a $x = 0$. On procède ensuite de la même manière que pour la détermination de la première donnée de test : la condition se réécrit en $(x < y) \wedge (x = 0) \wedge (y < 10)$ dont la donnée de test $(x = 0, y = 2)$ est une instance. La génération de données de test pour la condition de vérification $P_1 \wedge (\text{ si } P_2 \text{ alors } P_3 \text{ sinon } P_4)$ aboutit donc au jeu de test $\{(x = 2, y = 3), (x = 1, y = 20), (x = 0, y = 2)\}$.

La qualité des jeux de test obtenus par cette méthode découle donc directement de celle des heuristiques associées aux différents prédicats. On peut remarquer que le fait d'évaluer les conditions de vérification nécessite que le formalisme dans lequel est exprimée la spécification soit calculable. Le langage utilisé dans [CAR81] est une version typée du langage LISP.

Cette méthode propose une évaluation locale de la correction d'un programme. Ceci a l'avantage de rendre plus aisée la localisation des fautes. Par ailleurs, l'utilisation de spécifications formelles apporte une aide non négligeable pour déterminer si le résultat de l'exécution est correct ou pas (le problème de l'oracle est donc moins difficile).

3.2.2 L'analyse de partition

L'analyse de partition [RC85] consiste à comparer la spécification et l'implantation à tester afin de vérifier leur cohérence et d'engendrer des jeux de test. L'analyse de partition est une méthode générale qui est adaptable à n'importe quel langage de spécification à partir du moment où les descriptions peuvent être exprimées sous forme de domaines et de calculs associés.

Pour appliquer cette méthode, il faut tout d'abord dériver par évaluation symbolique des représentations fonctionnelles pour la spécification et pour l'implantation. Les représentations obtenues sont exprimées sous forme de fonctions partielles sur des sous-domaines du domaine d'entrée. En d'autres termes, la spécification et l'implantation sont définies chacune comme une union de fonctions opérant sur des sous-domaines du domaine d'entrée. Le principe de l'analyse de partition est de faire l'intersection des sous-domaines de la spécification avec les sous-domaines de l'implantation pour obtenir la partition de la procédure en sous-domaines qui sont traités uniformément. À chaque sous-domaine de la partition, on associe une description de ce sous-domaine, une description du calcul à effectuer selon la spécification et une description du calcul à effectuer selon l'implantation.

Ainsi, l'analyse de partition se décompose en quatre étapes :

1. Dérivation de représentations fonctionnelles pour la spécification et l'implantation.
2. Détermination de la partition de la procédure en utilisant diverses techniques d'analyse, en particulier l'évaluation symbolique.
3. Utilisation de la vérification pour montrer l'égalité des deux descriptions sur chaque sous-domaine.
4. Dérivation de données de test pour chaque sous-domaine, à l'aide d'informations issues de l'étape précédente.

• Détermination de la partition de la procédure

En premier lieu, il faut s'assurer de la compatibilité entre la spécification et l'implantation. On doit donc vérifier que leurs entrées et leurs sorties sont en mêmes nombres, de mêmes types et qu'elles prennent leurs valeurs dans un même domaine. En cas de forte incompatibilité (par exemple, des nombres de paramètres différents), l'analyse de

partition s'arrête. En revanche, s'il n'y a pas d'incompatibilité ou si celle-ci est mineure (cas de domaines de définition légèrement différents, en particulier à cause des différences existant entre le langage de spécification et celui d'implantation), l'analyse continue en n'opérant la phase de vérification que sur l'intersection des domaines et la phase de test sur l'union de ces domaines.

Pour cela, on construit la partition de la procédure en faisant l'intersection deux à deux des sous-domaines de la spécification avec les sous-domaines de l'implantation. On obtient ainsi les sous-domaines D constituant la partition et on associe à chaque sous-domaine D une expression C de la différence symbolique entre les descriptions du calcul à effectuer selon la spécification et l'implantation (ces descriptions sont des vecteurs d'expressions algébriques pour les paramètres de sortie, dérivés par évaluation symbolique).

On peut maintenant aborder les phases suivantes qui s'appliquent indépendamment sur chaque sous-domaine de la partition.

- **Phase de vérification**

Il s'agit de vérifier que, sur chaque sous-domaine D de la partition, la spécification et l'implantation effectuent le même calcul. Pour cela, on doit montrer que la différence de calcul C associée est nulle, en utilisant des techniques de preuve standards telles que celles présentes dans les vérificateurs automatiques. Souvent, la simplification de C conduit effectivement à la valeur zéro, mais parfois il est nécessaire de décomposer D en sous-domaines et de prouver que C est nulle sur chacun de ces sous-domaines de façon indépendante. Lorsque cela se produit, la décomposition obtenue est réutilisée afin de diriger le test conduit dans la phase suivante.

- **Phase de test**

On peut s'interroger sur l'intérêt du test lorsqu'il intervient suite à une phase de vérification. Il y a en fait deux situations possibles. Soit la vérification échoue : l'intérêt de la phase de test est donc évident. Soit la vérification réussit à établir un résultat : le test permet alors de récuser ou corroborer des hypothèses quant à l'environnement d'exécution, utilisées dans la phase de vérification. La phase de test n'est par conséquent jamais inutile. Comme lors de la phase précédente, les sous-domaines de la partition sont testés indépendamment les uns des autres. De manière générale, on peut classer les erreurs en deux catégories : les erreurs dans le calcul (le chemin traversé lors de l'exécution du programme testé correspond à celui attendu mais la sortie obtenue est incorrecte) et les erreurs de domaine (le chemin traversé n'est pas celui désiré). Pour chaque sous-domaine issu de la décomposition réalisée dans la phase de vérification, des données de test sont produites de façon à couvrir ces deux classes d'erreurs ; elles permettent en particulier de détecter les erreurs de chemin manquant (ceci étant rendu possible par l'utilisation d'une spécification).

Une évaluation de l'analyse de partition (utilisant l'analyse de mutation) montre que cette méthode fournit de bons résultats [RC85]. Cela provient surtout du fait qu'on manipule de l'information sur les comportements souhaités et effectifs du programme, obtenue par intersection de la spécification et de l'implantation [GOU83, WO80]. La

décomposition en sous-problèmes proposée par cette méthode est aussi intéressante puisqu'elle simplifie la tâche à accomplir en permettant de prouver et de tester le programme par parties.

Par contre, l'analyse de partition effectue beaucoup d'approximations car elle met en jeu des problèmes indécidables en général. En pratique, on peut aussi noter qu'un travail non négligeable sur le programme et la spécification sera nécessaire dans la plupart des cas avant de pouvoir appliquer la méthode. De plus, cette méthode n'est pas véritablement automatisée : c'est à l'utilisateur de produire les données de test à partir des sous-domaines déterminés lors de la phase de vérification.

3.3 Récapitulatif

Il ressort de cette étude que toutes les techniques de génération de jeux de test, qu'elles soient statiques ou dynamiques, uniquement basées sur des critères de test structurels ou intégrées dans des méthodes combinant la preuve et le test de programme, sont construites selon le schéma général représenté dans la Figure 3.1.

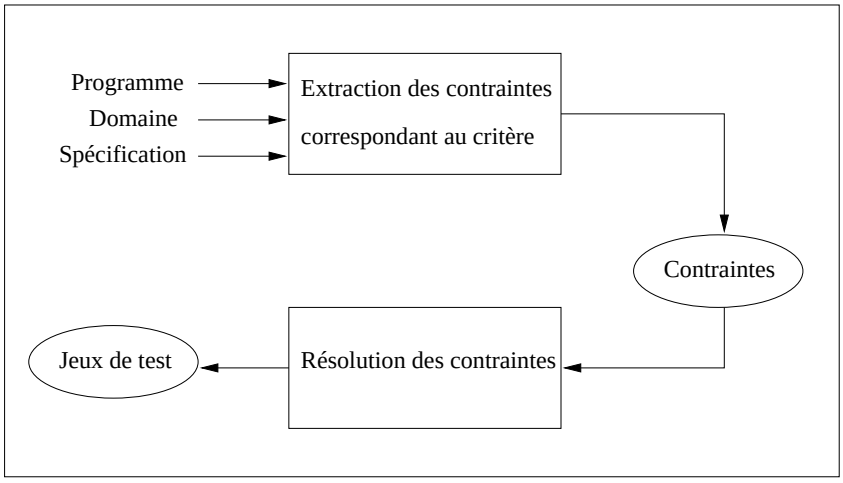


FIG. 3.1 – *Schéma général d'un générateur de jeux de test.*

Elles sont donc constituées de deux phases principales :

1. L'extraction des contraintes correspondant au critère.
2. La résolution des contraintes.

En général, la première phase est réalisée par une analyse statique du code (mais aussi de la spécification dans le cas où la méthode utilisée n'est pas purement structurelle) et les contraintes sont obtenues par évaluation symbolique. Cette phase est entièrement automatisée dans les techniques présentées ici, mis à part le cas de l'analyse de mutation. Cependant, des approximations importantes doivent être faites pour l'analyse statique des procédures, des boucles et des structures de données dynamiques.

La deuxième phase correspond à la production proprement dite des données de test. Pour ce faire, il existe plusieurs stratégies très différentes allant du simple recours

à l'utilisateur [OW91, RC85] jusqu'à une large automatisation grâce à des résolveurs de contraintes [DO91] en passant par l'utilisation de démonstrateurs de théorèmes interactifs [CAR81]. On remarque le recours presque systématique aux résolveurs de contraintes lorsque l'on effectue du test basé sur le flot de contrôle. On peut qualifier de statiques ces différentes façons de résoudre les contraintes puisqu'elles ne nécessitent pas d'exécuter le programme. Rappelons qu'il existe aussi des techniques de génération de jeux de test reposant sur une résolution dynamique des contraintes [KOR90].

Dans la pratique actuelle, ce sont les critères de test des instructions et de test des branches qui sont majoritairement utilisés pour engendrer des jeux test (mais aussi pour évaluer leur adéquation). En l'absence d'outils automatiques, ce sont en effet les critères qui sont le plus facilement applicables manuellement. On constate ainsi le fossé existant entre la théorie du test, qui propose des critères de plus en plus sophistiqués, et la pratique du test, qui souffre d'un très grand manque d'automatisation et ne peut donc utiliser les meilleurs critères.

Chapitre 4

Bilan

Deux types de démarches bien distinctes ressortent des chapitres précédents :

1. Des contributions de nature théorique qui conduisent à des définitions formelles de critères, de qualité, d'adéquation, d'hypothèses de test. Cependant, ces définitions formelles ne sont pas véritablement effectives en pratique pour différentes raisons :
   - D'une part, on se confronte en général à des problèmes d'indécidabilité (chemin impossible, validité d'une association définition/utilisation, équivalence de mutants, vérification d'une hypothèse de test...).
   - D'autre part, pour les critères purement structurels, on ne dispose pas d'une propriété à satisfaire. Faute d'objectif formel précis, il est difficile de décider entre des critères comme le test des branches et le test des associations définition/p-utilisation (*all-p-uses*). Quelle est, par exemple, la garantie formelle apportée par le fait qu'un jeu de test parcourt toutes les associations définition/utilisation ?
2. Des travaux de nature expérimentale, qui conduisent le plus souvent à des outils exploitant les critères a posteriori : dans ce cas, ils ne fournissent pas de véritable aide à la génération de jeux de test. Par ailleurs, le taux de couverture d'un critère ne pouvant pas être de 100% en général (à cause de l'existence de chemins impossibles), on ne peut pas dire qu'un jeu de test apporte une quelconque certitude quant au bon fonctionnement du programme. Les rares outils utilisant les critères a priori (effectuant donc une génération de jeux de test) souffrent des mêmes problèmes d'indécidabilité et d'incertitude quant à la qualité des données engendrées.

Ces démarches très différentes correspondent à des courants de travaux qui se sont relativement peu influencés mutuellement, ce qui se traduit par l'absence de méthodes de test effectives reposant sur des bases formelles. Cette situation est bien entendu dommageable ; nous ne prétendons pas y remédier dans cette thèse mais nous proposons d'aborder le problème de manière modeste en considérant des formalismes restreints pour la définition des programmes et des propriétés. Nous pensons en effet qu'un processus de génération de jeux de test bien fondé doit s'appuyer à la fois sur le programme à tester et sur une propriété que ce dernier est censé vérifier (l'objectif de test). Nous

constatons aussi que la plupart des difficultés rencontrées dans la génération de jeux de test (aussi bien en théorie qu'en pratique) proviennent de la puissance d'expression des langages utilisés (et de la perte inévitable de décidabilité qui s'ensuit). Pour échapper à cette fatalité, nous proposons d'utiliser des formalismes restreints (définis en l'occurrence par des schémas ou modèles de fonctions) qui nous permettent de générer automatiquement des jeux de test robustes[1] pour certaines classes de propriétés. Cette démarche fait l'objet de la seconde partie du document.

1. Nous utiliserons par la suite le terme de *complets* pour caractériser plus particulièrement les jeux de test produits selon notre méthode.

Deuxième partie

Génération de jeux de test pour des schémas de programmes et de propriétés

Chapitre 1

Démarche

La partie précédente nous a amenés à distinguer, dans le domaine du test, deux grandes catégories de travaux qui restent pour l'instant largement déconnectées. La première conduit à des méthodes de test utilisables en pratique mais n'apportant aucune garantie formelle (cf. Chapitre 2 de la Partie I : critère de test des branches, critère de test des associations définition/utilisation...). La seconde porte sur des caractérisations théoriques trop générales pour avoir un impact sur des méthodes de test effectives (cf. Section 2.4.1 de la Partie I : notions de fiabilité, validité, robustesse...).

On peut en fait décrire le test comme la confrontation du comportement effectif d'un programme à son comportement attendu (qui peut être donné par une spécification) pour une collection de données d'entrée. Trois types de documents sont donc mis en jeu : le programme, les données d'entrée et les propriétés. L'objectif de cette thèse est double : d'une part, nous voulons établir un lien formel entre ces trois artefacts ; d'autre part nous souhaitons pouvoir exploiter cette formalisation pour proposer une méthode effective permettant de déduire les données d'entrée à partir d'un programme et d'une propriété. Comme nous l'avons vu dans la partie précédente, la source principale de difficultés pour atteindre cet objectif est le degré de généralité des langages et formalismes considérés. La clé de notre démarche consiste donc à circonscrire la classe de programmes et la classe de propriétés considérées. Pour cela, nous définissons en fait une hiérarchie de classes (liées par la relation d'inclusion) en s'inspirant de la caractérisation des fonctions primitives récursives.

Nous choisissons de prouver la correction partielle de programmes par le biais de test de propriétés. Pour cela, nous introduisons une hiérarchie de schémas (ou de classes) de fonctions récursives, et nous montrons qu'un jeu de test fini complet $J(S)$ peut être associé à chaque schéma S. Le jeu de test $J(S)$ est dit complet relativement à S s'il est suffisant pour distinguer deux fonctions quelconques de S :

$$\forall f \in S \,.\, \forall g \in S \,.\, (f \neq g \Rightarrow \exists x \in J(S) \,.\, f(x) \neq g(x))$$

Un jeu de test complet permet donc de tester de manière sûre l'égalité de deux fonctions, sous réserve qu'elles appartiennent à la même classe. Étant donné que notre hiérarchie est ordonnée, un jeu de test $J(S)$ complet relativement à $S = Lub(S_1, S_2)$

est aussi complet relativement à S_1 et S_2. On peut ainsi déterminer l'égalité de deux fonctions appartenant à des schémas différents d'une même hiérarchie.

Le résultat précédent peut être mis à profit dans le contexte du test de la manière suivante. Supposons que nous voulons vérifier qu'un programme $Prog$ satisfait une propriété $\mathcal{P}_a$. $\mathcal{P}_a$ doit être exprimée sous forme de fonction sur un domaine abstrait $\mathcal{D}_a$. Les éléments de $\mathcal{D}_a$ sont des propriétés des arguments (ou des résultats) d'un programme. On choisira par exemple $\mathcal{D}_a = \mathbb{N}$, ensemble des entiers naturels, si on s'intéresse à des propriétés sur les longueurs de listes. Un programme $Prog$ satisfait une propriété $p_a \in \mathcal{D}_a \to \mathcal{D}_a$ s'il rend, pour tout argument satisfaisant $a \in \mathcal{D}_a$, un résultat satisfaisant $p_a(a)$. Par exemple, toujours avec $\mathcal{D}_a = \mathbb{N}$, la propriété $p_a(n) = 2 * n$ est satisfaite par tout programme qui rend une liste dont la taille est le double de celle de son argument. La première étape de la vérification consiste à dériver une version abstraite $Prog_a$ de $Prog$, ce qui peut être fait de manière formelle dans le cadre de l'interprétation abstraite. $Prog_a$ et $\mathcal{P}_a$ sont alors deux fonctions définies sur le même domaine abstrait $\mathcal{D}_a$. Une analyse syntaxique simple de ces fonctions (dans le style d'une inférence de type) nous permet de leur associer les schémas S_{Prog_a} et $S_{\mathcal{P}_a}$ dans la hiérarchie. La plus petite borne supérieure S de S_{Prog_a} et $S_{\mathcal{P}_a}$ est le plus petit schéma de la hiérarchie contenant à la fois $Prog_a$ et $\mathcal{P}_a$. D'après le résultat précédent, il est alors suffisant de tester $Prog_a$ sur un jeu de test complet $J(S)$ associé à S pour décider si $Prog_a$ satisfait ou non la propriété $\mathcal{P}_a$. Pour se ramener au test d'un programme initial $Prog$, on peut appliquer aux éléments de $J(S)$ une fonction de concrétisation (duale de la fonction d'abstraction utilisée pour obtenir $Prog_a$ à partir de $Prog$).

Nous prenons maintenant un exemple simple pour illustrer cette démarche. Considérons un programme naïf de renversement de listes (*Reverse*), écrit dans un langage fonctionnel du premier ordre.

$$\begin{array}{lcl} Reverse(nil) & = & nil \\ Reverse(n{:}l) & = & Append(Reverse(l), n) \\ \\ Append(nil, m) & = & m{:}nil \\ Append(n{:}l, m) & = & n{:}Append(l, m) \end{array}$$

Plutôt que d'essayer de prouver que ce programme est correct par rapport à une spécification complète de *Reverse* (tâche généralement hors de portée d'un outil automatique pour un programme et une spécification quelconques), nous aimerions pouvoir vérifier automatiquement qu'il satisfait certaines propriétés spécifiques. Par exemple, une propriété qu'un programme *Reverse* doit satisfaire est que la longueur de son résultat doit être égale à la longueur de son argument. Pour pouvoir vérifier cette propriété, nous devons l'exprimer comme une fonction calculant la *longueur attendue* du résultat de *Reverse* à partir de la longueur de son argument. Ici, cette fonction est l'identité.

L'étape suivante consiste à dériver une version abstraite de *Reverse*, calculant la *longueur effective* de son résultat à partir de la longueur de son argument. Cette dérivation peut être réalisée automatiquement en appliquant la technique de l'interprétation abstraite [CC77, CC92]. Une interprétation abstraite est définie par un domaine abstrait, une correspondance entre les domaines abstrait et standard (ou concret) et une

interprétation de toutes les primitives de base du langage dans le domaine abstrait. Si les interprétations de toutes les primitives de base sont correctes par rapport à la relation de correspondance entre les domaines, alors l'interprétation abstraite de tout programme du langage est aussi correcte. Dans notre cas, on abstrait les listes par des entiers naturels ; le domaine abstrait est donc le domaine des entiers naturels $\mathbb{N}$ et la relation de correspondance associe les listes avec leur longueur. Les valeurs autres que les listes sont abstraites sur un domaine ne contenant qu'une valeur, car elles ne sont pas utiles pour l'analyse considérée ici. Au lieu de garder ces arguments factices, nous décidons de les faire disparaître lors de l'abstraction. Ici, les primitives d'intérêt sont la fonction *cons* (représentée par «:» dans notre langage de programmation) et la constante *nil*. Leurs abstractions sont, respectivement, la fonction successeur $Succ = \lambda x.\ (x+1)$ et la constante 0. Dans ce cadre, nous obtenons l'interprétation abstraite suivante *Lreverse* (Longueur de *Reverse*) pour la fonction *Reverse*[1].

$$\begin{array}{lcl} Lreverse(0) & = & 0 \\ Lreverse(n+1) & = & Lappend(Lreverse(n)) \end{array}$$

$$\begin{array}{lcl} Lappend(0) & = & 1+0 \\ Lappend(n+1) & = & 1+Lappend(n) \end{array}$$

Il nous reste maintenant à comparer *Lreverse* avec la fonction identité. Dans ce cas simple, nous pourrions utiliser des manipulations symboliques et des techniques de preuves par récurrence pour montrer que *Lappend* est équivalente à la fonction *Succ*, et la remplacer ensuite par celle-ci dans le corps de *Lreverse*. Mais il est bien connu que la mécanisation de ces techniques est difficile en général. Au lieu de cela, nous analysons les définitions pour en dériver un jeu de test complet permettant de décider de leur équivalence (ou fournissant un contre-exemple si elles s'avèrent différentes). Le but de cette analyse statique simple (appelée *inférence de schéma* dans le reste du document) est d'identifier le schéma (ou squelette) des fonctions et de trouver leur position dans une hiérarchie de schémas. Pour le moment, considérons une hiérarchie simple de fonctions unaires définie de la manière suivante :

$$\begin{array}{lcl} S_1^1 & = & \{\lambda x.\ (x+k) \mid k \in \mathbb{N}\} \bigcup \{\lambda x.\ k \mid k \in \mathbb{N}\} \\ S_{i+1}^1 & = & \{f \mid f(0) = k \\ & & \qquad f(n+1) = g(f(n))\ ,\ k \in \mathbb{N},\ g \in S_i^1\} \end{array}$$

Un résultat de la Partie 2.2.2.3 montre que, quel que soit n, deux fonctions (distinctes) de S_n^1 diffèrent nécessairement sur au moins un des $n+1$ premiers entiers naturels. Plus formellement :

$$\forall f \in S_n^1\ .\ \forall g \in S_n^1\ .\ (f \neq g \Rightarrow \exists x \in \{0, \ldots, n\}\ .\ f(x) \neq g(x))$$

ce qui est équivalent à :

1. Notons que le second argument de *Append* n'est pas de type liste, ce qui explique pourquoi *Lappend* n'a qu'un seul argument

$$\forall f \in S_n^1 \,.\, \forall g \in S_n^1 \,.\, ((\forall x \in \{0, \ldots, n\} \,.\, f(x) = g(x)) \Rightarrow f = g)$$

Autrement dit, nous pouvons décider de l'égalité de deux fonctions appartenant à la classe S_n^1 en les testant uniquement sur les $n+1$ premiers entiers naturels.

L'inférence de schéma est réalisée par appariement («*pattern matching*») sur la structure de la définition des fonctions. Par exemple, la définition de *Lreverse* peut être appariée au motif générique des schémas S_n^1 :

$$\begin{array}{lcl} f(0) & = & k \\ f(n+1) & = & g(f(n)) \end{array}$$

avec $k = 0$ et $g = \mathit{Lappend}$. La définition de *Lappend* peut être appariée au motif générique avec $k = (1 + 0) = 0$ et $g = \lambda x.\ (1 + x)$. Ensuite, g peut être appariée avec *Succ*, qui appartient au schéma S_1^1. Ainsi, *Lappend* est associée au schéma S_2^1 et *Lreverse* est associée[2] au schéma S_3^1.

Il n'est pas difficile de montrer que les schémas S_n^1 définissent une hiérarchie de schémas qui est strictement croissante au sens de l'inclusion ensembliste (autrement dit, $n < n' \Rightarrow S_n^1 \subset S_{n'}^1$). *Id* appartenant au schéma S_1^1, nous devons donc prendre la plus petite borne supérieure de S_1^1 et S_3^1, qui est S_3^1. Le résultat mentionné précédemment nous permet alors de conclure qu'il est suffisant de tester *Id* et *Lreverse* sur les valeurs 0, 1, 2 et 3 pour décider de leur égalité. Pour exprimer ces valeurs dans les termes du programme original, nous avons juste à utiliser la relation de correspondance entre les domaines abstraits et concrets. Ici, cela signifie qu'il est suffisant de tester le programme *Reverse* sur quatre listes quelconques de longueurs 0, 1, 2 et 3 pour décider si *Reverse* rend toujours une liste de même longueur que son argument.

Pour résumer, les quatre principales étapes du processus de dérivation de jeux de test sont les suivantes :

Abstraction du programme : Programme → Programme abstrait

↓

Inférence de schéma : Programme abstrait → Schéma$_1$
Propriété → Schéma$_2$

↓

Inférence du jeu de test abstrait : *Lub*(Schéma$_1$,Schéma$_2$) → Jeu de test abstrait

↓

Génération du jeu de test concret : Jeu de test abstrait → Jeu de test concret

2. Étant donné que *Id* et *Lreverse* sont toutes deux égales (d'un point de vue sémantique) à la fonction identité, on aurait pu penser qu'elles auraient été tout simplement associées au schéma S_1^1, mais nous devons garder en mémoire que la connaissance de cette égalité sémantique n'est pas disponible à ce stade ; en fait, c'est exactement ce que nous essayons de prouver.

Nous nous intéressons donc à prouver par le test qu'un programme vérifie une propriété donnée. Pour cela, nous exploitons les résultats que nous avons obtenus pour des schémas de programmes sur les entiers. Ces résultats montrent l'existence d'un jeu de test fini complet permettant d'identifier sans ambiguïté une fonction parmi toutes celles qui appartiennent au même schéma (association d'un schéma et d'un jeu de test complet). Nous expliquons ces résultats dans le Chapitre 2, l'ensemble des programmes traités étant défini par une hiérarchie de schémas de programmes récursifs sur les entiers naturels. Comme nous l'avons décrit plus haut, la démarche que nous proposons consiste à écrire la propriété qui nous intéresse dans le moule des schémas de fonctions sur les entiers, puis à faire une interprétation abstraite du programme sur le domaine des entiers, en accord avec la propriété. Nous détaillons la procédure d'abstraction de programmes dans le Chapitre 3. Dans un second temps, il faut identifier les schémas de fonctions qui contiennent la propriété et le programme abstrait. Ceci est réalisé par l'utilisation de l'algorithme d'inférence de schéma que nous proposons dans le Chapitre 4. À ce stade, montrer que le programme vérifie la propriété équivaut à montrer que le programme abstrait et la propriété sont identiques. Nous nous retrouvons ainsi dans le cas des schémas sur les entiers, et nous pouvons déterminer un jeu de test abstrait complet fini permettant de juger si le programme vérifie la propriété. Une dernière étape, facultative, est de concrétiser le jeu de test abstrait. L'intérêt de cette phase est de ramener le problème à un test du programme initial. Nous donnons une procédure de concrétisation de jeu de test dans le Chapitre 5. Enfin, nous terminons par des exemples sur lesquels nous appliquons notre méthode (Chapitre 6).

Chapitre 2

Jeux de test complets pour des schémas de fonctions sur les entiers

La première étape de la méthode proposée dans le chapitre précédent consiste à définir des hiérarchies de schémas de fonctions récursives sur $\mathbb{N}$ et $\mathbb{N} \times \mathbb{N}$. Les résultats que nous obtenons dans ce cadre restreint nous permettent d'exhiber un jeu de test fini complet permettant de distinguer deux fonctions appartenant à des schémas quelconques d'une hiérarchie.

Pour donner l'intuition sous-jacente à l'obtention de ces résultats, nous étudions d'abord les propriétés des fonctions appartenant à une hiérarchie de schémas récursifs unaires simples sur les entiers (Partie 2.1). Nous montrons ensuite comment ces observations nous conduisent à proposer un jeu de test complet. Les Parties 2.2 et 2.3 présentent respectivement les résultats obtenus pour des schémas unaires plus complexes et des schémas binaires.

2.1 Intuition

Pour illustrer notre démarche, considérons d'abord l'ensemble (ou schéma) suivant de fonctions unaires sur les entiers :

$$S_1^1 \quad = \quad \{\lambda x.\ (x + k) \mid k \in \mathbb{N}\} \bigcup \{\lambda x.\ k \mid k \in \mathbb{N}\}$$

Cet ensemble regroupe les fonctions de base qui serviront à construire les schémas utilisés par la suite.

Nous cherchons à caractériser les fonctions de ce schéma afin d'en déduire un jeu de test permettant de les distinguer deux à deux. Pour ce faire, nous observons d'abord que ce schéma est l'union de deux ensembles disjoints de fonctions sur $\mathbb{N}$. D'une part, l'ensemble des fonctions constantes, et d'autre part, un sous-ensemble des fonctions affines caractérisé par l'équation $f(x) = x + k$ (notons que l'identité fait partie de cet ensemble). Ces deux ensembles sont aisés à tester indépendamment. Pour tester

l'égalité de deux fonctions constantes, il suffit d'une valeur de test quelconque. Dans le cas du sous-ensemble des fonctions affines, il suffit là aussi d'une valeur quelconque. Par contre, une donnée de test unique n'est pas suffisante pour distinguer deux fonctions appartenant à l'union de ces deux ensembles. En effet, quelle que soit la valeur d de test choisie, on ne peut pas distinguer la fonction constante $\lambda x.\ d$ de l'identité. Il est donc nécessaire de prendre une seconde valeur de test différente de la première. Le jeu de test ainsi obtenu, deux valeurs distinctes quelconques, est complet au sens où il permet de distinguer deux fonctions quelconques du schéma S_1^1. Notons qu'il permet aussi d'identifier sans ambiguïté toute fonction du schéma.

En fait, tout se passe comme si on avait besoin d'une donnée de test pour tester à l'intérieur de chaque sous-ensemble de la partition, et d'une autre donnée de test pour choisir entre les deux sous-ensembles. Le jeu de test rendu par notre méthode dans ce cas est composé des valeurs 0 et 1.

Intéressons nous maintenant à la hiérarchie définie par la règle de récurrence suivante :

$$\begin{aligned} S_{i+1}^1 \quad = \quad & \{f \mid f(0) = k \\ & \qquad f(n+1) = g(f(n))\ ,\ k \in \mathbb{N},\ g \in S_i^1\} \end{aligned}$$

On peut montrer facilement que l'ensemble des fonctions décrit par un schéma est inclus dans celui du schéma suivant dans la hiérarchie ($S_i^1 \subset S_{i+1}^1$).

Le schéma suivant S_1^1 dans la hiérarchie est donc :

$$\begin{aligned} S_2^1 \quad = \quad & \{f \mid f(0) = k \\ & \qquad f(n+1) = g(f(n))\ ,\ k \in \mathbb{N},\ g \in S_1^1\} \end{aligned}$$

et il peut aussi s'exprimer de la manière suivante :

$$\begin{aligned} S_2^1 \quad = \quad & \{\lambda x.\ k \mid k \in \mathbb{N}\} \\ \bigcup \quad & \{\lambda x.\ si\ \ x = 0\ \ alors\ \ k_1\ \ sinon\ \ k_2 \mid k_1 \neq k_2 \in \mathbb{N}\} \\ \bigcup \quad & \{\lambda x.\ k_1 + k_2 x \mid k_1,\ k_2 \in \mathbb{N},\ k_2 \neq 0\} \end{aligned}$$

Ce schéma est donc l'union de plusieurs ensembles disjoints de fonctions sur $\mathbb{N}$. Tout d'abord, l'ensemble des fonctions constantes déjà présentes dans le schéma S_1^1. Ensuite, l'ensemble des fonctions constantes à partir de la valeur 1 (et prenant une valeur quelconque en 0), et enfin un sous-ensemble de fonctions affines strictement plus grand que celui du schéma S_1^1. On peut en fait généraliser la notion de fonctions «constantes» et regrouper dans cette classe toutes les fonctions constantes à partir d'un seuil (ici 1). On se retrouve alors dans la même situation que pour le schéma précédent, avec des fonctions pseudo-constantes et affines. Pour tester les fonctions pseudo-constantes, deux valeurs de test sont nécessaires, l'une devant être égale à 0 et l'autre quelconque mais différente de 0. Pour tester les fonctions affines, on doit choisir deux valeurs quelconques. Cependant, deux valeurs ne suffisent pas pour tester le schéma dans sa globalité. Deux fonctions différentes, l'une affine et l'autre pseudo-constante, peuvent être égales sur 0 et 1 par exemple. À titre d'illustration, la fonction pseudo-constante

$$\lambda x.\ si\ x = 0\ alors\ k_1\ sinon\ (k_1 + k_2)$$

et la fonction affine

$$\lambda x.\ si\ x = 0\ alors\ k_1\ sinon\ (k_1 + k_2 x)$$

rendent toutes deux k_1 et $(k_1 + k_2)$ pour les valeurs 0 et 1, mais sont différentes pour les valeurs suivantes (si $k_2 \neq 0$). On peut montrer qu'ajouter au jeu de test une troisième valeur suffit pour distinguer les fonctions pseudo-constantes des fonctions affines de S_2^1. La méthode que nous proposons indique effectivement le jeu de test $\{0, 1, 2\}$ pour ce schéma.

Pour comprendre plus précisément la technique que nous proposons, examinons maintenant le schéma S_3^1. On a :

$$\begin{array}{rcl} S_3^1 & = & \{f \mid f(0) = k \\ & & \qquad f(n+1) = g(f(n))\ ,\ k \in \mathbb{N},\ g \in S_2^1\} \end{array}$$

En considérant les différents cas possibles pour $g \in S_2^1$, on peut réécrire S_3^1 de la manière suivante :

$$\begin{array}{rcl} S_3^1 & = & \{\lambda x.\ k \mid k \in \mathbb{N}\} \\ & \bigcup & \{\lambda x.\ si\ \ x = 0\ \ alors\ \ k_1\ \ sinon\ \ k_2 \mid k_1 \neq k_2 \in \mathbb{N}\} \\ & \bigcup & \{\lambda x.\ si\ \ x = 0\ \ alors\ \ k_1\ \ sinon\ \ si\ \ x = 1\ \ alors\ \ k_2\ \ sinon\ \ k_3 \\ & & \qquad\qquad\qquad\qquad \mid k_2 \neq k_3 \in \mathbb{N}\} \\ & \bigcup & \{\lambda x.\ si\ \ (x\ mod\ 2) = 0\ \ alors\ \ k_1\ \ sinon\ \ k_2 \mid k_1 \neq k_2 \in \mathbb{N}\} \\ & \bigcup & \{\lambda x.\ si\ \ x = 0\ \ alors\ \ k_1\ \ sinon\ \ si\ \ (x\ mod\ 2) = 0\ \ alors\ \ k_2\ \ sinon\ \ k_3 \\ & & \qquad\qquad\qquad\qquad \mid k_1,\ k_2,\ k_3 \in \mathbb{N},\ k_1 \neq k_3,\ k_2 \neq k_3\} \\ & \bigcup & \{\lambda x.\ k_1 k_3^x + k_2 k_3^{x-1} + \ldots + k_2 \mid k_1,\ k_2,\ k_3 \in \mathbb{N}\} \end{array}$$

où *mod* représente la fonction modulo.

Ce schéma est encore l'union de plusieurs ensembles disjoints de fonctions sur $\mathbb{N}$. On reconnaît l'ensemble des fonctions pseudo-constantes qui s'est agrandi (fonctions constantes à partir de 0, à partir de 1, ou à partir de 2). On découvre une nouvelle forme de fonctions : les fonctions périodiques à partir de 0 ou 1, et de période 2. Les fonctions pseudo-constantes peuvent être considérées comme des fonctions périodiques à partir de 0, 1 ou 2, et de période 1. Les fonctions pseudo-constantes et les fonctions périodiques présentent un comportement commun : elles prennent un nombre fini de valeurs distinctes et sont périodiques à partir d'un certain seuil (par la suite, nous qualifions ces fonctions de *δ-périodiques*). Le dernier ensemble de fonctions composant S_3^1 est un ensemble regroupant des fonctions exponentielles et des fonctions affines. Le comportement commun de ces fonctions est caractérisé par la notion d'*inflation*. Nous appelons *inflationnistes* les fonctions supérieures ou égales à l'identité. Tous les schémas que nous avons étudiés jusqu'à présent peuvent donc se décomposer en deux sous-ensembles disjoints : les fonctions *δ-périodiques* et les fonctions *inflationnistes*.

Suivons le même raisonnement que précédemment afin d'obtenir le jeu de test associé à S_3^1. Pour tester les fonctions *δ-périodiques*, on doit choisir un jeu de test contenant les valeurs 0, 1, 2 et 3. Par rapport au schéma précédent (où deux valeurs suffisaient à tester les fonctions pseudo-constantes), nous remarquons qu'il est nécessaire d'ajouter

deux nouvelles valeurs de test. Le premier ajout n'est pas surprenant car il correspond à la progression d'un niveau dans la hiérarchie. Par contre, le second est plus intriguant. Il est en fait nécessaire pour déterminer la période de la fonction (1 ou 2 dans le cas présent). Par exemple, la fonction $\lambda x.\ si\ x = 0\ alors\ k_1\ sinon\ \ si\ x = 1\ alors\ k_2\ sinon\ k_3$ de période 1 et la fonction $\lambda x.\ si\ x = 0\ alors\ k_1\ sinon\ \ si\ (x\ mod\ 2) = 0\ alors\ k_3\ sinon\ k_2$ de période 2 rendent toutes deux k_1, k_2 et k_3 pour les valeurs 0, 1 et 2, mais sont différentes pour les valeurs impaires suivantes (si $k_2 \neq k_3$). Les tester sur la valeur 3 permet donc de les distinguer. Dans le schéma précédent, la période valait toujours 1. Ce schéma ne reflétait donc pas toutes les potentialités des fonctions périodiques. En général, les fonctions périodiques d'un schéma peuvent avoir différentes périodes possibles, dépendant du niveau du schéma dans la hiérarchie. Pour le test des fonctions *inflationnistes*, il faut 3 valeurs distinctes quelconques. Pour distinguer deux fonctions du schéma S_3^1, on peut donc choisir le jeu de test suivant : $\{0, 1, 2, 3\}$. Il s'avère ainsi qu'à partir du rang 3, le jeu de test pour les fonctions *δ-périodiques* est suffisant pour tester l'ensemble du schéma.

Cette esquisse de l'étude de la hiérarchie des schémas de programmes S_n^1 nous permet d'introduire les principales idées qui sous-tendent notre méthode :

- chaque schéma de la hiérarchie se décompose en deux classes de fonctions : les fonctions *δ-périodiques* et les fonctions *inflationnistes* ;
- le nombre et les valeurs des données de test nécessaires au test de la classe des fonctions *δ-périodiques* dépend du nombre de valeurs distinctes que peut prendre la fonction avant de devenir périodique, qui dépend lui-même du niveau du schéma considéré dans la hiérarchie ;
- le nombre de données de test nécessaires au test de la classe des fonctions *inflationnistes* découle du niveau du schéma dans la hiérarchie (nous verrons par la suite que la détermination formelle de ce jeu de test nécessite l'introduction d'une nouvelle notion : la *α-séparabilité*) ;
- un jeu de test complet fini associé au schéma S_n^1 se déduit des deux jeux de test associés aux deux classes précédentes ;
- le jeu de test associé au plus grand des schémas de deux fonctions dans la hiérarchie (c'est-à-dire celui qui contient l'autre) est un jeu de test complet fini permettant de distinguer ces deux fonctions.

2.2 Résultats pour des schémas unaires

Les notions de base nécessaires à la définition formelle de la méthode sont introduites dans la Partie 2.2.1. Les Parties 2.2.2 et 2.2.3 présentent respectivement les résultats obtenus pour des schémas unaires simples puis pour des schémas unaires enrichis.

2.2.1 Définitions formelles

Nous avons vu dans la partie précédente que les fonctions décrites par un schéma pouvaient présenter deux comportements bien distincts. Elles peuvent prendre un nombre

fini de valeurs différentes en étant périodiques à partir d'un certain point (c'est-à-dire *δ-périodiques*), ou être supérieures ou égales à l'identité (c'est-à-dire *inflationnistes*). Nous donnons maintenant les définitions formelles correspondant à ces notions. Nous rappelons que tous les schémas considérés dans ce document caractérisent des fonctions sur les entiers. Le fait que les fonctions sont définies sur l'ensemble des entiers naturels $\mathbb{N}$ est souvent utilisé dans les preuves que nous présentons par la suite (en particulier, on a la propriété : $\forall n \in \mathbb{N},\ \forall n' \in \mathbb{N}\ .\ n > n' \Rightarrow n \geq n' + 1$).

Tout d'abord, nous introduisons la notion de fonction (λ, π)-périodique (Définition 1 et Propriété 1), que nous utilisons ensuite pour définir celle, plus générale, de fonction δ-périodique.

Définition 1
Une fonction f est dite (λ, π)-périodique, avec $\pi \geq 1$, si et seulement si

$$\begin{array}{ll} & \forall x \geq \lambda\ .\ \forall y \geq \lambda\ .\ (x\ mod\ \pi = y\ mod\ \pi \Rightarrow f(x) = f(y)) \\ \wedge & \forall x < \lambda + \pi\ .\ \forall y < \lambda + \pi\ .\ (f(x) = f(y) \Rightarrow x = y) \end{array}$$

où mod est la fonction modulo.
Remarque : *$(\lambda, 0)$-périodique est indéfini.*

Une fonction (λ, π)-périodique est donc une fonction donnant $(\lambda + \pi)$ résultats différents sur les $(\lambda + \pi)$ premiers entiers naturels, et devenant périodique de période π à partir de la valeur λ. La Figure 2.1 montre un exemple schématique de fonction (λ, π)-périodique[1].

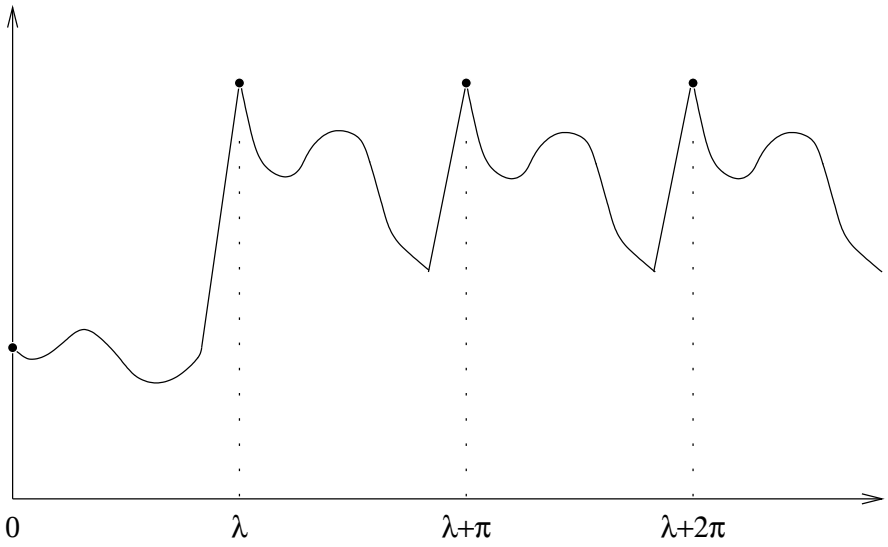


FIG. 2.1 – *Une fonction (λ, π)-périodique*

1. Pour une meilleure lisibilité, nous avons représenté la fonction par une courbe continue. Il faut cependant garder à l'esprit qu'il s'agit ici de fonctions sur les entiers.

Propriété 1
Une fonction f est (λ, π)-périodique, avec $\pi \geq 1$, si et seulement s'il existe une fonction g définie sur l'intervalle $[0, \pi - 1]$ telle que

$$\forall x \geq \lambda \ . \ f(x) = g(x \ mod \ \pi)$$

et, par ailleurs,

$$\forall x < \lambda + \pi \ . \ \forall y < \lambda + \pi \ . \ (f(x) = f(y) \Rightarrow x = y).$$

Preuve
Le second terme de la conjonction est identique dans la Propriété 1 et la Définition 1. Il suffit donc de démontrer l'équivalence des premiers termes de la conjonction.

$\forall x \geq \lambda \ . \ \forall y \geq \lambda \ . \ (x \ mod \ \pi = y \ mod \ \pi \Rightarrow f(x) = f(y))$
$\Longleftrightarrow$
$\exists \pi$ valeurs $V_0, \ldots, V_{\pi-1}$ telles que

$$\forall x \geq \lambda \ . \ \forall y \geq \lambda \ . \ (x \ mod \ \pi = y \ mod \ \pi = i \Rightarrow f(x) = f(y) = V_i)$$

$\Longleftrightarrow$
Si g est définie par : $\forall i \in [0, \pi - 1] \ . \ g(i) = V_i$ alors

$$\forall x \geq \lambda \ . \ \forall y \geq \lambda \ . \ (x \ mod \ \pi = y \ mod \ \pi = i \Rightarrow f(x) = f(y) = g(i))$$

$\Longleftrightarrow$
$\exists g : [0, \pi - 1] \rightarrow \mathbb{N} \ . \ \forall x \geq \lambda \ . \ (x \ mod \ \pi = i \Rightarrow f(x) = g(i))$
$\Longleftrightarrow$
$\exists g : [0, \pi - 1] \rightarrow \mathbb{N} \ . \ \forall x \geq \lambda \ . \ f(x) = g(x \ mod \ \pi)$

Donc, f (λ, π)-périodique $\Longleftrightarrow$

$$\begin{array}{rl} & \exists g : [0, \pi - 1] \rightarrow \mathbb{N} \ . \ \forall x \geq \lambda \Rightarrow f(x) = g(x \ mod \ \pi) \\ \wedge & \forall x < \lambda + \pi \ . \ \forall y < \lambda + \pi \ . \ (f(x) = f(y) \Rightarrow x = y). \end{array}$$

□

Nous utilisons maintenant ces définition et propriété afin de caractériser la notion de fonction δ-périodique.

Définition 2
Une fonction f est dite δ-périodique si et seulement si elle est (λ, π)-périodique avec $\lambda + \pi \leq \delta$.
Un ensemble E de fonctions est dit δ-périodique si et seulement si, pour toute fonction f appartenant à E, f est δ-périodique.

Il s'agit d'une notion plus générale que celle de fonction (λ, π)-périodique au sens où les valeurs λ et π ne sont pas données explicitement mais approximées par un intervalle $(1 \leq \lambda + \pi \leq \delta)$.

À présent, nous introduisons la notion de fonction inflationniste.

Définition 3
Une fonction f est dite inflationniste de seuil θ_f si et seulement si

$$\forall x < \theta_f \ . \ f(x) = x$$

et

$$\forall x \geq \theta_f \ . \ f(x) > x.$$

Un ensemble E de fonctions est dit inflationniste si et seulement si, pour toute fonction f appartenant à E, f est inflationniste.

Par commodité, on généralisera cette propriété au cas de l'identité en prenant $\theta_f = \infty$.

La Figure 2.2 présente un exemple schématique de fonction inflationniste. Cette fonction est égale à l'identité jusqu'à la valeur $\theta - 1$ et lui est strictement supérieure ensuite.

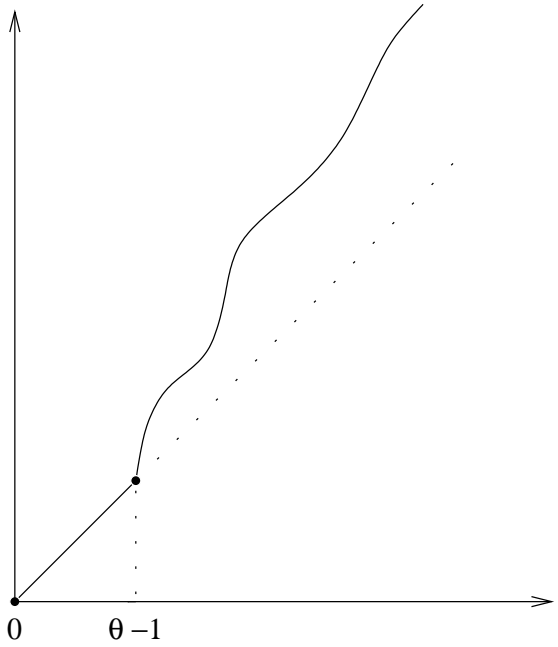


FIG. 2.2 – *Une fonction inflationniste de seuil θ*

Définition 4
Une fonction f est croissante si et seulement si $\forall x \ . \ \forall y \ . \ (x > y \Rightarrow f(x) > f(y))$.
Un ensemble E de fonctions est dit croissant si et seulement si, pour toute fonction f appartenant à E, f est croissante.

Propriété 2
Si f est croissante alors f est inflationniste.

Preuve
Supposons f croissante. Nous rappelons que f est une fonction de $\mathbb{N}$ dans $\mathbb{N}$.
• Nous prouvons d'abord par récurrence la propriété : $\forall x \in \mathbb{N} \,.\, f(x) \geq x$.

Cas de base : $f(0) \geq 0$ est trivialement vrai.
Cas de récurrence : supposons $f(x) \geq x$ et montrons $f(x+1) \geq x+1$.
$f(x+1) > f(x) \geq x$ car f est croissante et par hypothèse de récurrence.
Donc, $f(x+1) \geq x+1$.

• Pour prouver la Propriété 2, il faut montrer que :
– soit $\forall x \in \mathbb{N} \,.\, f(x) = x$, ce qui correspond à $\theta_f = \infty$;
– soit $\exists \theta_f \,.\, ((\forall x \geq \theta_f \,.\, f(x) > x) \wedge (\forall x < \theta_f \,.\, f(x) = x))$.

Si $\nexists x \,.\, f(x) > x$ alors $\forall x \in \mathbb{N} \,.\, f(x) = x$ car nous avons montré ci-dessus que $\forall x \in \mathbb{N} \,.\, f(x) \geq x$.

Si $\exists \theta_f \,.\, f(\theta_f) > \theta_f$ alors nous montrons par récurrence que $\forall x \geq \theta_f \,.\, f(x) > x$.

Cas de base : $f(\theta_f) > \theta_f$ est vrai.
Cas de récurrence : supposons $f(x) > x$ et montrons $f(x+1) > x+1$.
$f(x+1) > f(x) > x$ car f est croissante et par hypothèse de récurrence.
Donc, $f(x+1) > x+1$.
□

Notre objectif étant de distinguer deux fonctions quelconques d'un même ensemble, nous proposons la définition suivante de jeu de test complet.

Définition 5
J est un jeu de test complet pour une classe C de fonctions si et seulement si

$$\forall f \in C \,.\, \forall g \in C \,.\, (f \neq g \Rightarrow \exists x \in J \,.\, f(x) \neq g(x))$$

qui est équivalent à

$$\forall f \in C \,.\, \forall g \in C \,.\, ((\forall x \in J \,.\, f(x) = g(x)) \Rightarrow (f = g))$$

L'intérêt de la notion de δ-périodicité pour le test est exprimé par la propriété suivante, qui découle de la Définition 2.

Propriété 3
Si C est un ensemble de fonctions δ-périodiques alors $\{0, \ldots, \delta\}$ est un jeu de test complet pour C.

Preuve

Soit C un ensemble de fonctions δ-périodiques. Soient $f \in C$ et $g \in C$ telles que $\forall x \in \{0, \ldots, \delta\}$. $f(x) = g(x)$. Soient (λ_f, π_f) et (λ_g, π_g) les seuils et périodes respectifs de f et g avec $\lambda_f + \pi_f \leq \delta$ et $\lambda_g + \pi_g \leq \delta$.

Soit v la plus petite valeur de $\{0, \ldots, \delta\}$ telle que $\exists x \in [0, v-1]$. $f(x) = f(v)$. D'après la Définition 2, $v = \lambda_f + \pi_f$ et $x = \lambda_f$. Puisque f et g sont égales sur $\{0, \ldots, \delta\}$, v est aussi la plus petite valeur de $\{0, \ldots, \delta\}$ telle que $\exists x \in [0, v-1]$. $g(x) = g(v)$; on a donc aussi $v = \lambda_g + \pi_g$ et $x = \lambda_g$. On en conclut que $\lambda_f = \lambda_g$ et $\pi_f = \pi_g$.

On peut maintenant démontrer que $\forall x$. $f(x) = g(x)$.

• Si $x < \lambda_f$, $f(x) = g(x)$ puisque $\lambda_f \leq \delta$.

• Si $x \geq \lambda_f$, $\exists x' \in [\lambda_f, (\lambda_f + \pi_f - 1)]$. $x \ mod \ \pi_f = x' \ mod \ \pi_f$ (idem pour π_g). On a donc $f(x) = f(x')$ et $g(x) = g(x')$. Comme $x' \in \{0, \ldots, \delta\}$, on a $g(x') = f(x')$; on en conclut donc $f(x) = g(x)$.

Nous avons ainsi montré $(\forall x \in \{0, \ldots, \delta\} \ . \ f(x) = g(x)) \Rightarrow f = g$.

□

La détermination d'un jeu de test pour un ensemble de fonctions inflationnistes n'est pas aussi directe que dans le cas des fonctions δ-périodiques. Il nous est nécessaire d'introduire une notion intermédiaire importante: la *α-séparabilité*, illustrée par la Figure 2.3. Nous explicitons le lien entre les notions d'inflation et de α-séparabilité dans la Partie 2.2.2.2.

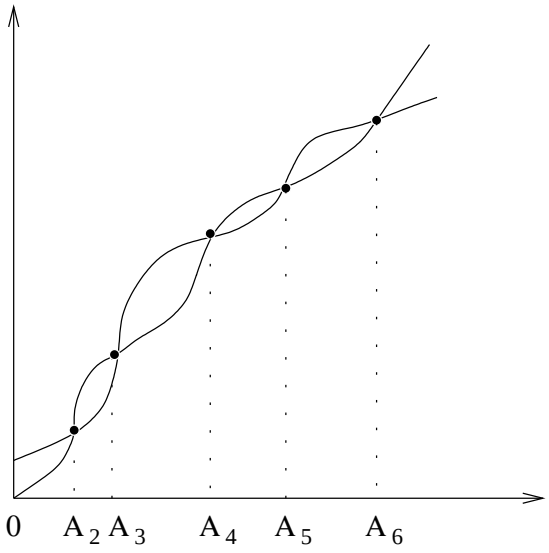


FIG. 2.3 – *Deux fonctions α-séparables (avec $\alpha \geq 6$)*

Définition 6

Deux fonctions f et g sont dites α-séparables si et seulement s'il existe a intervalles $I_1, \ldots, I_a$ avec $a \leq \alpha$, $I_i =]A_i, A_{i+1}[$, $A_1 = -1$, $A_{a+1} = \infty$, $i > j \Rightarrow A_i > A_j$, $p = pair$

ou $p = impair$, *et:*

$$\forall i \in [1,a] \;.\; (p(i) \Rightarrow (\forall x \in I_i \;.\; f(x) < g(x)) \wedge (A_{i+1} \neq \infty \Rightarrow f(A_{i+1}) \geq g(A_{i+1})))$$

$$\forall i \in [1,a] \;.\; (\neg p(i) \Rightarrow (\forall x \in I_i \;.\; f(x) > g(x)) \wedge (A_{i+1} \neq \infty \Rightarrow f(A_{i+1}) \leq g(A_{i+1})))$$

Un ensemble E de fonctions est dit α-séparable si et seulement si, pour toute fonction f appartenant à E, pour toute fonction g appartenant à E, f et g sont α-séparables.

Deux fonctions f et g sont α-séparables s'il est possible de décomposer l'ensemble des entiers naturels $\mathbb{N}$ en $a \leq \alpha$ intervalles $I_1, \ldots, I_a$ tels que l'une des deux fonctions est strictement plus grande (alternativement plus petite) que l'autre sur chaque intervalle. Dans la définition, a est la plus petite valeur satisfaisant cette propriété. On remarque que si deux fonctions sont α-séparables alors elles sont différentes. L'intérêt de la α-séparabilité pour le test vient de la Propriété 4 et du Corollaire 5, qui sont des conséquences directes de la Définition 6.

Propriété 4
Si f et g sont α-séparables et D un sous-ensemble de $\mathbb{N}$ tel que $\forall x \in D \;.\; f(x) = g(x)$ alors $Card(D) < \alpha$.

$Card(D)$ dénote le cardinal de l'ensemble D. La Propriété 4 implique que les fonctions α-séparables ne peuvent pas coïncider sur plus de $\alpha - 1$ valeurs. Autrement dit, tout jeu de test contenant au moins α valeurs est complet pour un ensemble de fonctions α-séparables. Notons que cette condition est moins contraignante que la propriété correspondante pour les fonctions δ-périodiques (Propriété 3) car cette dernière impose une condition sur le choix des valeurs de test : il faut choisir les $\delta + 1$ premiers entiers. Cela est dû au fait que les fonctions δ-périodiques peuvent exhiber un comportement spécifique sur les λ premiers entiers naturels (cf. Définition 2).

On déduit le Corollaire 5 suivant de la Propriété 4.

Corollaire 5
Si C est un ensemble de fonctions α-séparables alors $\{0, \ldots, \alpha - 1\}$ est un jeu de test complet pour C.

Preuve
Un jeu de test J est complet pour C (cf. Définition 5) s'il vérifie la propriété :

$$\forall f \in C \;.\; \forall g \in C \;.\; (f \neq g \Rightarrow \exists x \in J \;.\; f(x) \neq g(x))$$

Puisque $Card(\{0, \ldots, \alpha - 1\}) = \alpha$, la Propriété 4 implique bien :

$$\forall f \in C \;.\; \forall g \in C \;.\; (f \neq g \Rightarrow \exists x \in \{0, \ldots, \alpha - 1\} \;.\; f(x) \neq g(x))$$

□

Les résultats concernant la δ-périodicité et la α-séparabilité peuvent être réunis de la manière suivante :

Propriété 6
Si $E = E_1 \cup E_2$ avec E_1 croissant α-séparable et E_2 δ-périodique alors $\{0, \ldots, \mu\}$ est un jeu de test complet pour E avec $\mu = max(\alpha - 1, \delta)$.

Il s'avère que les fonctions inflationnistes apparaissant dans nos schémas sont également croissantes et α-séparables (cf. Propriété 13 et Propriété 16 de la Partie 2.2.2.2). Nous pouvons donc utiliser la Propriété 6 pour calculer les jeux de test complets associés aux schémas.

La Propriété 6 peut être prouvée en se basant sur la Propriété 3 et le Corollaire 5, et en montrant que les valeurs de test nécessaires pour distinguer deux fonctions de E_1 ou deux fonctions de E_2 sont suffisantes pour distinguer une fonction de E_1 et une fonction de E_2.

Preuve
Soit $E = E_1 \cup E_2$ avec E_1 croissant α-séparable et E_2 δ-périodique. Un jeu de test est complet pour E s'il permet de distinguer deux à deux toutes les fonctions de cet ensemble. Soient f et g deux fonctions quelconques de E. Pour montrer la Propriété 6, il faut donc différencier les cas possibles d'appartenances de f et g à E_1 et E_2. Si f et g appartiennent au même sous-ensemble, le Corollaire 5 (pour E_1) et la Propriété 3 (pour E_2) permettent de conclure. Autrement, l'une des fonctions est δ-périodique et l'autre croissante. D'après la Définition 2, on sait que le test d'une fonction δ-périodique sur la valeur δ rend une valeur déjà obtenue lors de son test sur les δ valeurs précédentes. Par ailleurs, on sait qu'une fonction croissante ne peut manifester un tel comportement puisqu'elle est nécessairement injective. Le jeu de test $\{0, \ldots, \delta\}$ permet donc de différencier une fonction δ-périodique d'une fonction croissante.

Plus formellement, d'après la Définition 5, J est un jeu de test complet pour E s'il vérifie la propriété :

$$\forall f \in E \,.\, \forall g \in E \,.\, (f \neq g \Rightarrow \exists x \in J \,.\, f(x) \neq g(x))$$

On doit donc montrer :

$$\forall f \in E_1 \cup E_2 \,.\, \forall g \in E_1 \cup E_2 \,.\, (f \neq g \Rightarrow \exists x \in J \,.\, f(x) \neq g(x))$$

C'est-à-dire

$$\forall f \in E_1 \,.\, \forall g \in E_1 \,.\, (f \neq g \Rightarrow \exists x \in J \,.\, f(x) \neq g(x))$$

$$\wedge$$

$$\forall f \in E_1 \,.\, \forall g \in E_2 \,.\, (f \neq g \Rightarrow \exists x \in J \,.\, f(x) \neq g(x))$$

$$\wedge$$

$$\forall f \in E_2 \,.\, \forall g \in E_2 \,.\, (f \neq g \Rightarrow \exists x \in J \,.\, f(x) \neq g(x))$$

Il est facile de vérifier que si on peut trouver J_1, J_2 et J_3 tels que :
- $\forall f \in E_1 \,.\, \forall g \in E_1 \,.\, (f \neq g \Rightarrow \exists x \in J_1 \,.\, f(x) \neq g(x))$,
- $\forall f \in E_2 \,.\, \forall g \in E_2 \,.\, (f \neq g \Rightarrow \exists x \in J_2 \,.\, f(x) \neq g(x))$,
- $\forall f \in E_1 \,.\, \forall g \in E_2 \,.\, (f \neq g \Rightarrow \exists x \in J_3 \,.\, f(x) \neq g(x))$.

alors $J = J_1 \cup J_2 \cup J_3$ est un jeu de test complet pour $E = E_1 \cup E_2$.

D'après le Corollaire 5 et la Propriété 3, il suffit de choisir $J_1 = \{0, \ldots, \alpha - 1\}$ et $J_2 = \{0, \ldots, \delta\}$.

D'après la Définition 2, si f est une fonction δ-périodique alors $\exists v \in [0, \delta - 1]$ tel que $f(\delta) = f(v)$. Par ailleurs, si g est une fonction croissante alors nous savons que $\forall x \in \mathbb{N} \,.\, \forall y \in \mathbb{N} \,.\, x \neq y \Rightarrow f(x) \neq f(y)$. Nous avons donc :

$$Card(\{f(x) \mid x \in [0, \delta]\}) = \delta + 1 \Rightarrow f \notin E_2$$

et

$$Card(\{f(x) \mid x \in [0, \delta]\}) \neq \delta + 1 \Rightarrow f \notin E_1$$

Donc, $\forall f \in E_1 \,.\, \forall g \in E_2 \,.\, \exists x \in \{0, \ldots, \delta\} \,.\, f(x) \neq g(x)$.

Le jeu de test $J_3 = J_2 = \{0, \ldots, \delta\}$ permet donc de différencier $f \in E_1$ de $g \in E_2$.

Ainsi, un jeu de test complet J pour E est :

$$J = J_1 \cup J_2 \cup J_3 = J_1 \cup J_2 = \{0, \ldots, \alpha - 1\} \cup \{0, \ldots, \delta\} = \{0, \ldots, \mu\}$$

avec $\mu = max(\alpha - 1, \delta)$.

□

Dans cette partie, nous avons vu que nous pouvions déterminer des jeux de test complets pour des ensembles de fonctions vérifiant certaines propriétés (δ-périodicité, croissance, α-séparabilité). Dans les parties suivantes (2.2.2 et 2.2.3), nous explorons différentes manières de construire des ensembles de fonctions vérifiant ces propriétés par le biais de schémas. Le premier schéma auquel nous nous intéressons permet de construire des fonctions récursives unaires sur les entiers naturels.

2.2.2 Schémas unaires simples

Nous commençons par rappeler la définition de la hiérarchie de schémas introduite dans la Partie 2.1.

Définition 7

$$\begin{array}{lcl} S_1^1 & = & \{\lambda x.\ (x + k) \mid k \in I\!N\} \bigcup \{\lambda x.\ k \mid k \in I\!N\} \\ S_{i+1}^1 & = & \{f \mid f(0) = k \\ & & \qquad f(n+1) = g(f(n)) \ ,\ k \in I\!N,\ g \in S_i^1\} \end{array}$$

Les schémas S_n^1 sont inspirés par des travaux sur les types inductifs et les schémas de programmes inductifs associés [PDM89]. Comme nous l'avons remarqué dans la Partie 2.1, la première observation à faire à propos des schémas S_n^1 est qu'ils peuvent être partitionnés en deux classes de fonctions : la première classe ne contient que des fonctions δ-périodiques, et la seconde ne contient que des fonctions inflationnistes. Nous les traitons dans cet ordre dans les Parties 2.2.2.1 et 2.2.2.2 avant de regrouper les résultats pour traiter les schémas complets dans la Partie 2.2.2.3.

2.2.2.1 Les fonctions δ-périodiques

Nous montrons d'abord un certain nombre de propriétés utiles satisfaites par les fonctions (λ, π)-périodiques. Ces propriétés sont essentiellement basées sur le fait que toute fonction f de S_n^1 peut s'écrire sous la forme : $f(x) = g^x(k)$. Elles nous permettent ensuite de prouver des résultats importants (Propriétés 11 et 12) sur la propagation de la δ-périodicité à travers la hiérarchie de schémas $\{S_n^1,\ n \in \mathbb{N}\}$.

Lemme 7
Si f (λ, π)-périodique alors $Card(f(\mathbb{N})) \leq \lambda + \pi$.

Preuve
Découle directement de la Propriété 1.
□

Propriété 8
Si f (λ, π)-périodique alors

$$\forall k\ .\ \exists j \in]1, (\lambda + \pi + 1)]\ .\ \exists j' \in [1, j[\ .\ f^j(k) = f^{j'}(k).$$

Preuve
Nous montrons la propriété en raisonnant par l'absurde.
Supposons f (λ, π)-périodique et $\exists k\ .\ \forall j \in]1, (\lambda + \pi + 1)]\ .\ \forall j' \in [1, j[\ .\ f^j(k) \neq f^{j'}(k)$.
On a alors : $\exists k\ .\ \forall i \in [1, (\lambda + \pi + 1)]\ .\ \forall i' \in [1, (\lambda + \pi + 1)]\ .\ i \neq i' \Rightarrow f^i(k) \neq f^{i'}(k)$.
On en conclut : $Card(f(\mathbb{N})) \geq (\lambda + \pi + 1)$, ce qui contredit le Lemme 7.
□

Propriété 9
Soient f (λ, π)-périodique et v, v' les deux plus petites valeurs de j, j' satisfaisant la Propriété 8 pour une valeur k donnée.
On a alors :

$$\forall \alpha \geq 0\ .\ \forall i \in [0, (v - v' - 1)]\ .\ f^{v'+i+\alpha(v-v')}(k) = f^{v'+i}(k).$$

Preuve
Soit $P(\alpha) \equiv (\forall i \in [0,(v-v'-1)]\ .\ f^{v'+i+\alpha(v-v')}(k) = f^{v'+i}(k))$.
Prouvons $P(n)$ par récurrence sur n.
• Cas de base: $P(0)$ est trivialement vraie.
• Cas de récurrence: supposons $P(n)$ vraie et montrons $P(n+1)$ vraie.
$P(n+1) \equiv (\forall i \in [0,(v-v'-1)]\ .\ f^{v'+i+(n+1)(v-v')}(k) = f^{v'+i}(k))$.
Or,
$f^{v'+i}(k) = f^{v'}(f^i(k)) = f^v(f^i(k)) = f^{v+i}(k) = f^{v+i+v'-v'}(k) = f^{v-v'}(f^{v'+i}(k))$.
Par hypothèse de récurrence ($P(n)$), on a :

$$\forall i \in [0,(v-v'-1)]\ .\ f^{v'+i+n(v-v')}(k) = f^{v'+i}(k).$$

Donc, $f^{v-v'}(f^{v'+i}(k)) = f^{v-v'}(f^{v'+i+n(v-v')}(k)) = f^{v'+i+(n+1)(v-v')}(k)$.
Nous concluons alors que: $f^{v'+i}(k) = f^{v'+i+(n+1)(v-v')}(k)$,
et donc $P(n+1)$ est vraie.
Nous avons ainsi montré: $\forall \alpha \geq 0\ .\ \forall i \in [0,(v-v'-1)]\ .\ f^{v'+i+\alpha(v-v')}(k) = f^{v'+i}(k)$.
□

Propriété 10
Soient f (λ,π)-périodique et v, v' les deux plus petites valeurs de j, j' satisfaisant la Propriété 8 pour une valeur k donnée.
Alors il existe une fonction h définie sur l'intervalle $[0, v-v'-1]$ telle que:

$$\forall x\ .\ (x \geq v' \Rightarrow f^x(k) = h(x\ mod\ (v-v'))).$$

Preuve
Il suffit de définir h par: $i \in [0,(v-v'-1)] \Rightarrow h(i) = f^{v'+i}(k)$.
La Propriété 9 implique: $\forall x \geq v'\ .\ f^x(k) = f^{v'+(x\ mod\ (v-v'))}(k)$.
On a donc bien: $\forall x \geq v'\ .\ f^x(k) = h(x\ mod\ (v-v'))$.
□

La propriété suivante nous permet de montrer que la périodicité se propage à travers la hiérarchie de schémas récursifs.

Propriété 11
Si g (λ,π)-périodique et f définie par:

$$\begin{array}{lcl} f(0) & = & k \\ f(n+1) & = & g(f(n)) \end{array}$$

alors f δ-périodique avec $\delta = (\lambda+\pi+1)$.

Preuve
Soient g (λ,π)-périodique et v, v' les deux plus petites valeurs de j, j' satisfaisant la Propriété 8 pour la valeur k utilisée dans la définition de $f(0)$ (ci-dessus). La Propriété 8 nous permet de conclure que:

$\forall 1 \leq i' < i < v \;.\; g^i(k) \neq g^{i'}(k)$
C'est-à-dire :
$\forall 1 \leq i' < i < v \;.\; f(i) \neq f(i')$.

Nous devons maintenant distinguer les deux cas suivants :

1. $\forall i \in [1, v[\;.\; f(i) \neq f(0)$
2. $\exists i \in [1, v[\;.\; f(i) = f(0)$

• Premier cas : $\forall i \in [1, v[\;.\; f(i) \neq f(0)$.
Dans ce cas, on peut conclure que :

$$\forall 0 \leq i' < i < v \;.\; f(i) \neq f(i'). \qquad (1)$$

D'autre part, d'après la Propriété 10, il existe une fonction h définie sur l'intervalle $[0, (v - v' - 1)]$ telle que :

$$\forall x \geq v' \;.\; g^x(k) = h(x \; mod \; (v - v'))$$

c'est-à-dire

$$\forall x \geq v' \;.\; f(x) = h(x \; mod \; (v - v'))$$

avec $1 \leq v' < v \leq (\lambda + \pi + 1)$.
Donc, en posant $n = v'$ et $m = v - v'$, on conclut qu'il existe une fonction h définie sur l'intervalle $[0, (m - 1)]$ telle que :

$$\forall x \geq n \;.\; f(x) = h(x \; mod \; m).$$

Par ailleurs, (1) implique :

$$\forall x < n + m \;.\; \forall y < n + m \;.\; (f(x) = f(y) \Rightarrow x = y).$$

Nous avons ainsi montré que, dans ce premier cas, f est (n, m)-périodique (cf. Propriété 1) avec $(n + m) = v \leq (\lambda + \pi + 1)$.

• Second cas : $\exists i \in [1, v[\;.\; f(i) = f(0)$.
Soit $m \in [1, v - 1]$ la plus petite valeur telle que $f(m) = f(0)$.
On a alors : $\forall 0 \leq i' < i < m \;.\; f(i) \neq f(i')$,
c'est-à-dire $\forall x < m \;.\; \forall y < m \;.\; (f(x) = f(y) \Rightarrow x = y)$.
En faisant un raisonnement similaire à celui du premier cas, on montre qu'il existe une fonction h définie sur l'intervalle $[0, (m - 1)]$ telle que :

$$\forall x \geq 0 \;.\; f(x) = h(x \; mod \; m).$$

Nous avons donc montré que, dans ce second cas, f est $(0, m)$-périodique (cf. Propriété 1) avec $m < v \leq (\lambda + \pi + 1)$.

En rassemblant les résultats des deux précédents cas selon la Définition 2, nous pouvons finalement conclure que f est δ-périodique avec $\delta = (\lambda + \pi + 1)$.

□

La propriété suivante caractérise la propagation de la propriété de périodicité à travers la hiérarchie de schémas et conclut notre traitement des fonctions périodiques. La Définition 2 et la Propriété 11 permettent de montrer la Propriété 12.

Propriété 12
Soit f définie par

$$\begin{array}{lcl} f(0) & = & k \\ f(n+1) & = & g(f(n) \end{array}$$

avec g δ-périodique,
alors f est $(\delta+1)$-périodique.

2.2.2.2 Les fonctions inflationnistes

Nous nous intéressons maintenant à la classe des fonctions inflationnistes engendrées par le schéma S_n^1. Le résultat suivant est utile pour montrer qu'un schéma ne peut contenir que des fonctions inflationnistes ou δ-périodiques.

Propriété 13
Soit f définie par

$$\begin{array}{lcl} f(0) & = & k \\ f(n+1) & = & g(f(n)) \end{array}$$

avec g inflationniste.
Si $k \geq \theta_g$ alors f est croissante et inflationniste.
Si $k < \theta_g$ alors f est δ-périodique (en fait, f est la fonction constante $f(n) = k$).

La seconde partie de la Propriété 13 vient du fait que les fonctions inflationnistes coïncident avec l'identité pour des valeurs inférieures à leur seuil (Définition 3). La première partie est prouvée par récurrence sur l'argument de f.

Preuve
• Premier cas : $k \geq \theta_g$.

Pour prouver que f est inflationniste, nous montrons par récurrence sur n les propriétés suivantes :

1. $\forall n \,.\, f(n) \geq \theta_g$
2. $\forall n \,.\, f(n+1) > f(n)$ (croissance de f)

(1) •• Cas de base : $f(0) = k \geq \theta_g$.
(1) •• Cas de récurrence : supposons $f(n) \geq \theta_g$.
$f(n+1) = g(f(n)) > f(n) \geq \theta_g$.
Donc, $\forall n \,.\, f(n) \geq \theta_g$.

(2) •• Cas de base : $f(1) = g(f(0)) = g(k) > k = f(0)$ car $k \geq \theta_g$.
(2) •• Cas de récurrence : supposons $f(n) > f(n-1)$.
$f(n+1) = g(f(n)) > f(n)$ car $f(n) \geq \theta_g$.
Donc, f est croissante.

La fonction f étant croissante, la Propriété 2 permet de conclure que f est inflationniste.

• Second cas : $k < \theta_g$.
On prouve par récurrence que $\forall n \; . \; f(n) = k$.
Le cas de base est direct ($f(0) = k$).
Supposons maintenant $f(n) = k$ et montrons $f(n+1) = k$.
Puisque $k < \theta_g$ on a $g(k) = k$ d'après la Définition 3.
Donc $f(n+1) = g(f(n)) = g(k) = k$.
□

On peut également associer à chaque schéma de la hiérarchie une borne supérieure des seuils des fonctions inflationnistes lui appartenant (Propriété 15). Mais auparavant, nous devons démontrer le lemme suivant (utile pour la preuve de la Propriété 15) :

Lemme 14
Si f est une fonction inflationniste appartenant à un schéma S_n^1 telle que $f \neq \lambda x. \; (x+k)$ et i un entier tel que $i \geq n$ alors on a : $f(i-1) - f(i-2) > 1$.

Preuve
Nous prouvons ce lemme par récurrence sur n.

Cas de base : le schéma S_1^1 ne contient pas de fonction inflationniste autre que $\lambda x. \; (x+k)$. Le lemme est donc trivialement vrai.

Cas de récurrence : supposons le lemme vrai pour n et montrons le pour $n+1$.
La fonction f est inflationniste et appartient au schéma S_{n+1}^1 donc il existe (cf. Définition 7 et Propriétés 12, 13 et 17) une fonction g inflationniste appartenant au schéma S_n^1 telle que :

$$f(i-1) = g(f(i-2))$$

et

$$f(i-2) = g(f(i-3)).$$

• Premier cas : supposons $g \neq \lambda x. \; (x+k)$.
Soit $K = f(i-2) + 1$.
La fonction f est inflationniste donc $f(i-2) \geq i-2$.
De plus, $i-2 \geq n-1$ car $i \geq n+1$ par hypothèse de récurrence (puisque $f \in S_{n+1}^1$).
Donc $f(i-2) \geq n-1$ et $K \geq n$.
On a également : $f(i-1) - f(i-2) = g(K-1) - g(f(i-3))$.

La fonction f est croissante (puisque inflationniste) donc $f(i-3) < f(i-2)$ et $f(i-3) \leq f(i-2) - 1 = K - 2$.

La fonction g étant croissante, on a aussi : $g(f(i-3)) \leq g(K-2)$.

Donc, $g(K-1) - g(f(i-3)) \geq g(K-1) - g(K-2) > 1$ par hypothèse de récurrence puisque $K \geq n$.

• Second cas : supposons $g = \lambda x.\ (x + k')$.

$k' \neq 0$ car si on avait $k' = 0$, on aurait $g = \lambda x.\ x$ et f serait une fonction constante, ce qui contredit l'hypothèse que f est inflationniste.

$k' \neq 1$ car sinon on aurait $g = \lambda x.\ (x+1)$ et donc $f = \lambda x.\ (x+k)$, ce qui contredit les hypothèses.

Donc, $k' > 1$ et $f(i-1) = g(f(i-2)) = f(i-2) + k' > f(i-2) + 1$.

□

Propriété 15

Si f est une fonction inflationniste différente de l'identité appartenant à un schéma S_n^1 alors $\theta_f \leq n - 1$.

Preuve

• Si $f = \lambda x.\ (x + k)$ avec $k \neq 0$ alors $\theta_f = 0$. Donc, $\forall n\ .\ \theta_f \leq n - 1$.

• Sinon, on peut appliquer le Lemme 14. On a :

$$\forall i \geq n\ .\ f(i-1) - f(i-2) > 1$$

donc, en particulier :

$$f(n-1) - f(n-2) > 1$$

c'est-à-dire :

$$f(n-1) > 1 + f(n-2) \geq n - 1$$

car $f(n-2) \geq n - 2$ puisque f est inflationniste.

Donc, $f(n-1) > n - 1$ et on a alors : $\theta_f \leq n - 1$.

□

Le résultat suivant correspond à la Propriété 12 pour la propagation de la propriété de α-séparabilité à travers la hiérarchie de schémas.

Propriété 16

Soient f et f' définies par

$$\begin{array}{lcl} f(0) & = & k \\ f(n+1) & = & g(f(n)) \end{array}$$

$$\begin{array}{lcl} f'(0) & = & k' \\ f'(n+1) & = & g'(f'(n)) \end{array}$$

avec g et g' α-séparables et inflationnistes, $k \geq \theta_g$ et $k' \geq \theta'_g$,
alors f et f' sont $(\alpha + 1)$-séparables.

Le fait que f et f' sont également inflationnistes a été établi par la Propriété 13. Le coeur de la preuve de la Propriété 16 consiste à associer à chaque intervalle J_j (à part le dernier) satisfaisant la Définition 6 pour f et f', un intervalle I_i satisfaisant la Définition 6 pour g et g'. Différents cas sont considérés, dépendants de la vacuité de J_j. Dans chaque cas, le fait que les fonctions g et g' sont inflationnistes joue un rôle crucial.

Preuve

Les fonctions f et f' de la Propriété 16 peuvent être définies de manière plus concise comme :

- $f(n) = g^n(k)$
- $f'(n) = g'^n(k')$

• Nous établissons d'abord que si g et g' sont α-séparables alors il existe β tel que f et f' sont β-séparables (c'est-à-dire qu'il existe un plus petit $b \leq \beta$ tel qu'il est possible de décomposer l'ensemble $\mathbb{N}$ en b intervalles sur lesquels l'une des deux fonctions est strictement plus grande, alternativement plus petite, que l'autre).

En effet, si g et g' sont α-séparables, il existe un plus petit a satisfaisant la Définition 6. D'après cette même définition, il existe donc aussi une valeur A_a telle que $\forall x > A_a \ . \ g(x) < g'(x)$ (ou l'inverse). Et puisque $k \geq \theta_g$, $k' \geq \theta'_g$, f et f' sont inflationnistes et croissantes (Propriété 13). Il existe donc c tel que $f(c) > A_a$ et $f'(c) > A_a$; puisque $f(c+m) = g^m(f(c))$ et $f'(c+m) = g'^m(f'(c))$, il existe u tel que $\forall c > u \ . \ f(c) < f'(c)$. On a donc $J_b =]B_b, \infty[$ avec B_b le plus petit u satisfaisant cette propriété. On peut ensuite découper l'intervalle $]-1, B_b[$ selon la Définition 6.

• Soient $J_j =]B_j, B_{j+1}[,\ j \in [1, b]$, les intervalles associés à f, f' par la Définition 6. On montre maintenant que $b \leq a+1$ en associant à chaque intervalle $J_j,\ j \in [1, b-1]$, un intervalle différent $I_i,\ i \in [1, a]$.

Soit $J_j =]B_j, B_{j+1}[,\ j > 1,\ B_{j+1} > B_j$, et $\forall n \in J_j \ . \ g^n(k) < g'^n(k')$ (l'inverse se traite de la même façon). On suppose $j \neq b$, donc $B_{j+1} \neq \infty$. D'après la Définition 6, on a également $g^{B_{j+1}}(k) \geq g'^{B_{j+1}}(k')$ et $g^{B_j}(k) \leq g'^{B_j}(k')$.

On distingue les cas $B_{j+1} > B_j + 1$ et $B_{j+1} = B_j + 1$.

•• Premier cas : $B_{j+1} > B_j + 1$.

Alors $J_j \neq \varnothing$ et $g^{(B_{j+1})-1}(k) < g'^{(B_{j+1})-1}(k')$.

La fonction g est croissante donc $g^{B_{j+1}}(k) < g(g'^{(B_{j+1})-1}(k'))$.

$g'^{B_{j+1}}(k') \leq g^{B_{j+1}}(k)$ donc :

$$g'(g'^{(B_{j+1})-1}(k')) < g(g'^{(B_{j+1})-1}(k')). \qquad (1)$$

•• Second cas : $B_{j+1} = B_j + 1$.

On a alors $g^{B_j}(k) \leq g'^{B_j}(k')$ et $g(g^{B_j}(k)) \geq g'(g'^{B_j}(k'))$.

La fonction g est croissante donc :

$$g(g'^{B_j}(k')) \geq g(g^{B_j}(k)) \geq g'(g'^{B_j}(k')). \qquad (2)$$

Des résultats (1) et (2), on déduit qu'il est possible d'associer à J_j une valeur v_j telle que $v_j \in H_j = [g'^{B_j}(k'), g'^{(B_{j+1})-1}(k')]$ et $g'(v_j) \leq g(v_j)$.

La fonction g' est inflationniste donc $j \neq j' \Rightarrow H_j \cap H_{j'} = \varnothing$.
Donc $j \neq j' \Rightarrow v_j \neq v_{j'}$. Donc tous les v_j sont différents et $j > j' \Rightarrow v_j > v_{j'}$.
Par ailleurs, si on a $g'(v_j) \leq g(v_j)$ alors $g'(v_{j+1}) \geq g(v_{j+1})$ (et vice versa).
Donc, si $I_i =]A_i, A_{i+1}[,\ i \in [1, a]$ sont les intervalles associés à $(g,\ g')$ alors il existe i et i' avec $i \neq i'$ tels que $v_j \in [A_i, A_{i+1}[$ et $v_{j+1} \in [A_{i'}, A_{i'+1}[$.
On a donc associé à chaque intervalle J_j (pour $j \neq b,\ j \neq 1$)un intervalle $[A_i, A_{i+1}[$ différent. Si $j = 1$, on a l'intervalle $]A_1, A_2[$. On a donc : $b \leq a + 1$.
Nous avons donc montré que f et f' sont β-séparables avec $\beta \leq \alpha + 1$. D'après la Définition 6, f et f' sont également $(\alpha + 1)$-séparables.
□

2.2.2.3 Dérivation de jeux de test complets pour les schémas S_n^1

Propriété 17
$\forall n\ .\ S_n^1 = E_n^1 \cup E_n^2$ *avec* E_n^1 *inflationniste n-séparable et* E_n^2 *n-périodique.*

Cette propriété est prouvée par récurrence sur n, en utilisant la Propriété 13, la Propriété 16 et la Propriété 12.

Preuve

• Cas de base : $S_1^1 = \{\lambda x.\ (x + k) \mid k \in \mathbb{N}\} \cup \{\lambda x.\ k \mid k \in \mathbb{N}\}$

L'ensemble de fonctions $\{\lambda x.\ (x + k) \mid k \in \mathbb{N}\}$ est 1-séparable puisque cet ensemble de fonctions est complètement ordonné. Par ailleurs, toute fonction $\lambda x.\ (x + k)$ est inflationniste (et de seuil 0 si $k > 0$, ∞ sinon).

D'autre part, l'ensemble de fonctions $\{\lambda x.\ k \mid k \in \mathbb{N}\}$ est 1-périodique.

• Cas de récurrence : supposons $S_n^1 = E_n^1 \cup E_n^2$ avec E_n^1 inflationniste n-séparable et E_n^2 n-périodique.

$$\begin{aligned} S_{n+1}^1 \quad = \quad & \{f \mid f(0) = k \\ & \qquad f(x+1) = g(f(x))\ ,\ k \in \mathbb{N},\ g \in S_n^1\} \end{aligned}$$

donc $S_{n+1}^1 = E_1' \cup E_2'$ avec :

$$\begin{aligned} E_1' \quad = \quad & \{f \mid f(0) = k \\ & \qquad f(x+1) = g(f(x))\ ,\ k \in \mathbb{N},\ g \in E_n^1\} \\ E_2' \quad = \quad & \{f \mid f(0) = k \\ & \qquad f(x+1) = g(f(x))\ ,\ k \in \mathbb{N},\ g \in E_n^2\} \end{aligned}$$

D'après les Propriétés 13 et 16, on a : $E_1' = E_1'' \cup \{\lambda x.\ k \mid k \in \mathbb{N}\}$ avec E_1'' inflationniste $(n + 1)$-séparable.

Par ailleurs, la Propriété 12 nous permet de conclure que E_2' est $(n+1)$-périodique. On a de plus $\{\lambda x.\ k \mid k \in \mathbb{N}\} \subset E_2'$ puisque la hiérarchie $\{S_n^1,\ n \in \mathbb{N}\}$ est ordonnée ; donc $S_{n+1}^1 = E_1'' \cup E_2'$ avec E_1'' inflationniste $(n+1)$-séparable et E_2' $(n+1)$-périodique.
□

La conjonction de la Propriété 17 et de la Propriété 6 nous permet de dériver la propriété suivante qui conclut notre traitement des schémas unaires simples.

Propriété 18
$\forall n$. $\{0, \ldots, n\}$ *est un jeu de test complet pour* S_n^1.

2.2.3 Schémas unaires plus complexes

La hiérarchie de schémas considérée dans la Partie 2.2.2 est limitée. Elle nous a surtout permis d'introduire les principes de base de notre méthode. Nous explorons maintenant une hiérarchie de schémas unaires plus riche, le cas des schémas binaires étant étudié dans la Partie 2.3. Pour chacun de ces nouveaux schémas, nous pouvons déterminer un jeu de test complet en réutilisant les résultats obtenus précédemment.

La hiérarchie $\{S_n^2,\ n \in \mathbb{N}\}$ que nous proposons est une généralisation de la hiérarchie $\{S_n^1,\ n \in \mathbb{N}\}$ permettant des appels récursifs qui ne s'appliquent pas directement à l'argument n :

Définition 8

$$\begin{array}{rcl} S_i^2 & = & \{f \mid f(0) = k \\ & & \quad f(n+1) = g(f(h(n)))\ ,\ k \in I\!N,\ g \in S_i^1,\ h \in S_i^1\} \end{array}$$

Une première différence par rapport aux schémas S_n^1 est que S_n^2 ne contient pas que des fonctions totales. Selon la valeur de la fonction h en effet, l'appel récursif à f peut s'appliquer à un argument supérieur à l'argument initial. Nous nous penchons donc d'abord sur le domaine de définition des fonctions de S_n^2.

Propriété 19
Soit f *définie par*

$$\begin{array}{rcl} f(0) & = & k \\ f(n+1) & = & g(f(h(n))) \end{array}$$

avec $g \in S_i^1$ *et* $h \in S_i^1$.
Si f *est définie pour toutes les valeurs de l'ensemble* $\{1, \ldots, i\}$ *alors* f *est totale.*

Preuve
Afin de montrer cette propriété, nous distinguons les cas h i-périodique et h inflationniste.

- Cas h i-périodique.
  Si h est i-périodique alors il existe $\pi \leq i$ tel que $\forall n$. $h(n+\pi) = h(n)$.
  On a donc :

$$\forall n\ .\ f(n+\pi+1) = g(f(h(n+\pi))) = g(f(h(n))) = f(n+1).$$

  Tester f sur les valeurs 1 à i est donc suffisant pour statuer sur sa (non-)terminaison.

- Cas h inflationniste.

Deux situations peuvent se présenter. Soit h est égale à l'identité, et alors la fonction f obtenue appartient à S^1_{i+1} (cf. Définition 7) ; soit h est différente de l'identité et on a : $\theta_h \leq i-1$ (d'après la Propriété 15, car $h \in S^1_i$). Dans ce dernier cas, f n'est pas totale ; de plus, sa non-terminaison se manifeste sur au moins une des valeurs de l'ensemble $\{1, \ldots, i\}$ puisque $h(\theta_h) > \theta_h$ avec $\theta_h < i$.
□

Lorsque h est i-périodique, la fonction f obtenue, quand elle termine, n'est ni périodique (au sens de la Définition 2) ni inflationniste. Elle a cependant un comportement cyclique (comme les fonctions périodiques) mais elle peut répéter certaines valeurs là où les fonctions périodiques garantissent des valeurs distinctes. Néanmoins, la propriété suivante nous permet de distinguer ce nouveau type de fonction des fonctions périodiques et inflationnistes rencontrées jusqu'ici.

Propriété 20
Soit f définie par

$$\begin{array}{lcl} f(0) & = & k \\ f(n+1) & = & g(f(h(n))) \end{array}$$

avec $g \in S^1_i$ et $h \in S^1_i$ i-périodique.
Si f est totale alors l'ensemble $\{0, \ldots, 2i\}$ permet d'identifier f.

Preuve
L'ensemble d'arrivée de f se calcule de la manière suivante :

$$\{f(x) \mid x \in \mathbb{N}\} = \{k\} \cup \{g(f(h(x))) \mid x \in \mathbb{N}\}.$$

Or, la fonction h est i-périodique donc :

$$Card(\{h(x) \mid x \in \mathbb{N}\}) \leq i.$$

Comme f est totale, on a alors :

$$Card(\{f(x) \mid x \in \mathbb{N}\}) \leq i+1.$$

Ainsi, la fonction f a un comportement cyclique (puisque son codomaine est fini).

Pour déterminer la période de f, il est nécessaire de reconnaître une séquence d'au moins i valeurs de $f(x)$ à partir de $x = i+1$. On considère

$$v_0 = f(0),\ v_1 = f(1),\ \ldots\ v_{2i} = f(2i).$$

On note $v_0 : \ldots : v_{2i}$ la séquence des $2i+1$ premières valeurs de f.

La fonction f étant cyclique, sa période est la plus petite valeur π telle qu'il existe une séquence $K = v_{2i-\pi+1} : \ldots : v_{2i}$ de valeurs, avec :

$$v_0 : \ldots : v_{2i} =< v_0 : \ldots : v_u >:: K :: \ldots :: K$$

avec au moins 2 occurrences de K et $u < i$.
□

En rassemblant les résultats obtenus dans cette partie, nous pouvons vérifier que : $\forall n \; . \; S^1_{n+1} \subset S^2_n$. Nous sommes maintenant en mesure de dériver un jeu de test complet pour les fonctions totales d'un schéma S^2_n.

Propriété 21
$\forall n \; . \; \{0, \ldots, 2n\}$ *est un jeu de test complet pour les fonctions totales de* S^2_n.

Preuve
Soit f définie par

$$\begin{array}{lcl} f(0) & = & k \\ f(x+1) & = & g(f(h(x))) \end{array}$$

avec $g \in S^1_n$ et $h \in S^1_n$.

D'après les résultats précédents, on a :

- si h est l'identité alors f est totale et $f \in S^1_{n+1}$ (cf. Définition 7),
- si h est i-périodique alors f est totale et on est dans le cas de la Propriété 20,
- si h est inflationniste mais différente de l'identité alors f n'est pas totale et sa non-terminaison est détectée par le jeu de test $\{1, \ldots, n\}$ (cf. Propriété 19).

D'après la Propriété 18, $\{0, \ldots, n+1\}$ est un jeu de test pour S^1_{n+1}.

D'autre part, $\{0, \ldots, 2n\}$ est un jeu de test pour les fonctions cycliques de S^2_n. De plus, il permet de distinguer ces fonctions des fonctions périodiques et inflationnistes de S^1_{n+1}.

Donc, $\{0, \ldots, 2n\}$ est un jeu de test complet pour les fonctions totales de S^2_n.
□

Toutes les fonctions considérées jusqu'ici sont unaires. Nous nous intéressons maintenant aux fonctions binaires (Partie 2.3).

2.3 Résultats pour des schémas binaires

Dans cette partie, nous montrons comment les résultats établis pour des schémas unaires dans la Partie 2.2 peuvent être utilisés afin de dériver des jeux de test complets pour des schémas binaires.

Dans un premier temps, nous définissons les schémas binaires auxquels nous nous intéressons (Partie 2.3.1). Nous proposons ensuite des jeux de test complets pour ces schémas dans la Partie 2.3.2.

2.3.1 Schémas binaires

Les schémas suivants capturent quelques définitions récursives communes :

Définition 9

$$\begin{aligned}
B^1(X_1, X_2) &= \{f \mid f(0,m) = h(m) \\
&\qquad f(n+1,m) = g(f(n,m)) \ , \ g \in X_1, \ h \in X_2\} \\
\\
B^2(X_1, X_2) &= \{f \mid f(0,m) = h(m) \\
&\qquad f(n+1,m) = f(n,g(m)) \ , \ h \in X_1, \ g \in X_2\} \\
\\
B^3(X) &= \{f \mid f(0,m) = k \\
&\qquad f(n+1,m) = g(m,f(n,m)) \ , \ k \in I\!N, \ g \in X\}
\end{aligned}$$

Ainsi, les schémas binaires B^1 et B^2 sont paramétrés par les schémas unaires associés aux fonctions les définissant, et le schéma binaire B^3 est paramétré par un des schémas binaires de la définition. La Propriété 17 montre que chaque schéma unaire S_i^1 se partitionne en deux classes de fonctions correspondant aux fonctions δ-périodiques et aux fonctions inflationnistes. Afin de pouvoir traiter les schémas binaires, nous devons considérer ces deux classes de fonctions séparément. Nous les appelons P_i et I_i respectivement (pour δ-périodiques et inflationnistes).

Définition 10

$$\begin{aligned}
P_1 &= \{\lambda x.\ k \mid k \in I\!N\} \\
P_{i+1} &= \{f \mid f(0) = k \\
&\qquad f(n+1) = g(f(n)) \ , \ k \in I\!N, \ g \in P_i\} \\
\\
I_1 &= \{\lambda x.\ (x+k) \mid k \in I\!N\} \\
I_{i+1} &= \{f \mid f(0) = k \\
&\qquad f(n+1) = g(f(n)) \ , \ k \geq i-1, \ g \in I'_i\} \\
\\
I'_1 &= \{\lambda x.\ (x+k) \mid k > 0\} \\
I'_{i+1} &= \{f \mid f(0) = k \\
&\qquad f(n+1) = g(f(n)) \ , \ k \neq 0, \ g \in I'_i\} \\
&\bigcup \{f \mid f(0) = 0 \\
&\qquad f(n+1) = g(f(n)) \ , \ g \in I''_i\} \\
\\
I''_1 &= \{\lambda x.\ (x+k) \mid k > 1\} \\
I''_{i+1} &= \{f \mid f(0) = k \\
&\qquad f(n+1) = g(f(n)) \ , \ k > 1, \ g \in I'_i\} \\
&\bigcup \{f \mid f(0) = 0 \\
&\qquad f(n+1) = g(f(n)) \ , \ g \in I''_i\} \\
&\bigcup \{f \mid f(0) = 1 \\
&\qquad f(n+1) = g(f(n)) \ , \ g \in I''_i\}
\end{aligned}$$

La sous-classe de fonctions I_i' est utilisée afin d'exclure la possibilité que $g = id$ dans la définition de I_{i+1} (cf. Propriétés 13 et 15). Le schéma intermédiaire I_i'' permet de supprimer de manière syntaxique les différentes définitions possibles pour la fonction identité et la fonction successeur.

Nous pouvons maintenant établir les principaux résultats concernant les schémas binaires.

2.3.2 Dérivation de jeux de test complets pour les schémas binaires

Les propriétés suivantes permettent d'associer un jeu de test complet à la plupart des schémas paramétrés de la Définition 9. Nous portons d'abord notre attention sur la preuve de ces propriétés avant de discuter leur portée. Les preuves présentées dans cette partie sont plus «opérationnelles» que celles de la Partie 2.2.2. Cela est dû au fait que nous ne cherchons pas à prouver qu'un schéma vérifie certaines propriétés comme précédemment, mais seulement qu'il est possible de le tester. La première propriété concerne les schémas $B^1(X_1, X_2)$.

Propriété 22
$(\{0\} \times \{0, \ldots, j-1\}) \cup (\{1\} \times \{0, \ldots, i-1\})$ *est un jeu de test complet pour* $B^1(I_i, I_j)$.
$\{0, \ldots, i\} \times \{0, \ldots, j\}$ *est un jeu de test complet pour* $B^1(I_i, P_j)$.
$\{0, \ldots, i+1\} \times \{0, \ldots, j\}$ *est un jeu de test complet pour* $B^1(P_i, P_j)$.

Preuve
Nous considérons successivement les trois cas de la Propriété 22 pour les valeurs des paramètres de B^1.

• **Schéma** $B^1(I_i, I_j)$

Soient f et f' appartenant à $B^1(I_i, I_j)$ et

$$\forall (x, y) \in (\{0\} \times \{0, \ldots, j-1\}) \cup (\{1\} \times \{0, \ldots, i-1\}) \ . \ f(x, y) = f'(x, y).$$

On a donc

$$\begin{array}{lcl} f(0, m) & = & h(m) \\ f(n+1, m) & = & g(f(n, m)) \end{array}$$

et

$$\begin{array}{lcl} f'(0, m) & = & h'(m) \\ f'(n+1, m) & = & g'(f'(n, m)) \end{array}$$

avec $g \in I_i$, $g' \in I_i$, $h \in I_j$ et $h' \in I_j$.

La propriété

$$\forall (x, y) \in \{0\} \times \{0, \ldots, j-1\} \ . \ f(x, y) = f'(x, y)$$

est équivalente à

$$\forall y \in \{0, \ldots, j-1\} \ . \ h(y) = h'(y)$$

ce qui implique $h = h'$ d'après les Propriétés 4 et 17 (puisque $h \in I_j$ et $h' \in I_j$).

La propriété

$$\forall (x, y) \in \{1\} \times \{0, \ldots, i-1\} \ . \ f(x, y) = f'(x, y)$$

est équivalente à

$$\forall y \in \{0, \ldots, i-1\} \ . \ g(h(y)) = g'(h'(y)) = g'(h(y)).$$

Comme h est croissante, tous les $h(y)$ sont différents et g et g' sont égales sur un domaine de cardinalité i. De nouveau, la Propriété 4 nous permet de conclure $g = g'$. De $h = h'$ et $g = g'$, nous pouvons déduire $f = f'$.

• **Schéma** $B^1(I_i, P_j)$

Soient f et f' appartenant à $B^1(I_i, P_j)$ et

$$\forall (x, y) \in \{0, \ldots, i\} \times \{0, \ldots, j\} \ . \ f(x, y) = f'(x, y).$$

On a donc

$$\begin{array}{lcl} f(0, m) & = & h(m) \\ f(n+1, m) & = & g(f(n, m)) \end{array}$$

et

$$\begin{array}{lcl} f'(0, m) & = & h'(m) \\ f'(n+1, m) & = & g'(f'(n, m)) \end{array}$$

avec $g \in I_i$, $g' \in I_i$, $h \in P_j$ et $h' \in P_j$.

La propriété

$$\forall (x, y) \in \{0\} \times \{0, \ldots, j\} \ . \ f(x, y) = f'(x, y)$$

est équivalente à

$$\forall y \in \{0, \ldots, j\} \ . \ h(y) = h'(y)$$

ce qui implique $h = h'$ d'après la Propriété 3.

Pour y fixé, les fonctions

$$f_{h(y)}(x) = f(x, y) = g^x(h(y))$$

et

$$f'_{h(y)}(x) = f'(x,y) = g'^{x}(h(y)) \qquad (1)$$

sont inflationnistes (et éléments de S^1_{i+1}) ou constantes d'après la Propriété 13.

Pour y quelconque, la propriété

$$\forall x \in \{0,\ldots,i\} \;.\; f(x,y) = f'(x,y)$$

implique donc

$$\forall x \in \mathbb{N} \;.\; f_{h(y)}(x) = f'_{h(y)}(x) \qquad (2)$$

d'après la Propriété 4.

Par ailleurs, h étant j-périodique, son codomaine est couvert par les arguments $\{0,\ldots,j-1\}$; plus précisément :

$$\{h(n) \mid n \in \mathbb{N}\} = \{h(n) \mid n \in \{0,\ldots,j-1\}\}. \qquad (3)$$

Les résultats (1), (2) et (3) permettent donc de montrer que

$$\forall (x,y) \in \{0,\ldots,i\} \times \{0,\ldots,j-1\} \;.\; f(x,y) = f'(x,y)$$

implique

$$\forall (x,y) \in \mathbb{N} \times \mathbb{N} \;.\; f(x,y) = f'(x,y).$$

• **Schéma** $B^1(P_i, P_j)$

Soient f et f' appartenant à $B^1(P_i, P_j)$ et

$$\forall (x,y) \in \{0,\ldots,i+1\} \times \{0,\ldots,j\} \;.\; f(x,y) = f'(x,y).$$

On a donc

$$\begin{array}{lcl} f(0,m) & = & h(m) \\ f(n+1,m) & = & g(f(n,m)) \end{array}$$

et

$$\begin{array}{lcl} f'(0,m) & = & h'(m) \\ f'(n+1,m) & = & g'(f'(n,m)) \end{array}$$

avec $g \in P_i$, $g' \in P_i$, $h \in P_j$ et $h' \in P_j$.

La propriété

$$\forall (x,y) \in \{0\} \times \{0,\ldots,j\} \;.\; f(x,y) = f'(x,y)$$

est équivalente à

$$\forall y \in \{0, \ldots, j\} \; . \; h(y) = h'(y)$$

ce qui implique $h = h'$ d'après la Propriété 3.

Pour y fixé, les fonctions

$$f_{h(y)}(x) = f(x, y) = g^x(h(y))$$

et

$$f'_{h(y)}(x) = f'(x, y) = g'^x(h(y)) \qquad (1)$$

sont $(i+1)$-périodiques d'après la Propriété 12.

Pour y quelconque, la propriété

$$\forall x \in \{0, \ldots, i+1\} \; . \; f(x, y) = f'(x, y)$$

implique donc

$$\forall x \in \mathbb{N} \; . \; f_{h(y)}(x) = f'_{h(y)}(x) \qquad (2)$$

d'après la Propriété 3.

Par ailleurs, h étant j-périodique, son codomaine est couvert par les arguments $\{0, \ldots, j-1\}$; plus précisément :

$$\{h(n) \mid n \in \mathbb{N}\} = \{h(n) \mid n \in \{0, \ldots, j-1\}\}. \qquad (3)$$

Les résultats (1), (2) et (3) permettent donc de montrer que

$$\forall (x, y) \in \{0, \ldots, i+1\} \times \{0, \ldots, j-1\} \; . \; f(x, y) = f'(x, y)$$

implique

$$\forall (x, y) \in \mathbb{N} \times \mathbb{N} \; . \; f(x, y) = f'(x, y).$$

□

Les propriétés suivantes établissent le même type de résultats que la Propriété 22 pour les schémas $B^2(X_1, X_2)$ et $B^3(X)$ respectivement.

Propriété 23
$(\{0\} \times \{0, \ldots, i-1\}) \cup (\{1\} \times \{0, \ldots, j-1\})$ *est un jeu de test complet pour* $B^2(I_i, I_j)$.
$(\{0\} \times \{0, \ldots, i-1\}) \cup (\{1\} \times \{0, \ldots, j\})$ *est un jeu de test complet pour* $B^2(I_i, P_j)$.

Propriété 24
$\{(0,0)\} \cup (\{1\} \times \{1, \ldots, i\}) \cup (\{1, \ldots, j\} \times \{1\})$ *est un jeu de test complet pour* $B^3(B^1(I_i, I_j))$.

Les preuves de ces propriétés sont données dans l'annexe C. Elles sont de nature similaire à celle de la Propriété 22.

Notons que nous avons proposé des jeux de test complets pour les instanciations les plus communes des schémas binaires. Celles qui n'ont pas été traitées correspondent à des combinaisons improbables dans les programmes réels. Par exemple, le schéma $B^1(P_i, I_j)$ est défini par :

$$\begin{aligned} B^1(P_i, I_j) \quad = \quad & \{f \mid f(0,m) = h(m) \\ & \qquad f(n+1,m) = g(f(n,m)) \ , \ g \in P_i, \ h \in I_j\} \end{aligned}$$

Il s'agit donc de l'application récursive d'une fonction périodique avec comme cas terminal un appel à une fonction croissante.

Il est intéressant de remarquer que ces combinaisons improbables conduisent aussi à des schémas de programmes difficiles à tester. Pour le schéma $B^1(P_i, I_j)$ par exemple, la difficulté provient du fait qu'il n'y a aucun moyen d'assurer qu'un nombre fini d'arguments de la fonction inflationniste h produise un ensemble de résultats couvrant le domaine de g (modulo sa périodicité).

Chapitre 3

Abstraction

Nous avons montré dans le chapitre précédent comment établir l'égalité de deux fonctions à partir d'un ensemble fini de données de test. Notre but est d'exploiter ce résultat pour montrer automatiquement qu'un programme satisfait une propriété donnée. Pour ce faire, nous devons exprimer à la fois la *propriété escomptée* et la *propriété effective* du programme comme des fonctions dans un même domaine, avant de les comparer pour établir leur égalité. La *propriété escomptée* est spécifiée par le programmeur (ou le testeur). L'interprétation abstraite [CC77, CC92] est une technique permettant d'extraire des propriétés d'un programme. Dans ce chapitre, nous utilisons cette technique pour réaliser l'abstraction du programme initial, qui permet d'obtenir la *propriété effective*. Nous explicitons les principes de l'interprétation abstraite dans la Partie 3.1. Comme précédemment dans ce document, nous illustrons notre méthode sur des programmes manipulant des listes, et nous considérons des propriétés s'exprimant en termes de longueur de listes. Dans ce cas, nous pouvons exprimer l'interprétation abstraite par transformation de programmes (Partie 3.2).

Avant d'approfondir la procédure d'abstraction, nous présentons les langages L_{Prog} et L_{Prop} que nous utilisons pour exprimer les programmes sources et les propriétés.

Le langage L_{Prog} est un langage fonctionnel typé du premier ordre, permettant de définir des fonctions non-récursives unaires et des fonctions récursives unaires ou binaires. Le langage manipule deux «types» de données : les éléments d'un type de base (entiers naturels, réels...) et les listes[1]. Les différents cas des définitions récursives sont déterminés par appariement («*pattern matching*») avec 0 ou *nil* comme cas de base. Chaque définition de variable est accompagnée de son type. Par la suite, nous faisons l'hypothèse que les programmes manipulés sont bien typés.

1. Nous considérons que les types des éléments d'une liste peuvent être quelconques.

Langage de programmation L_{Prog} :

$$
\begin{array}{lcl}
Prog & = & Def\ ;\ Prog\ \mid \\
 & & Def. \\
Def & = & Let\ Fun\ (Var{:}Type,\ Exp)\ \mid \\
 & & Letrec\ Fun\ Case(0,\ Exp,\ Add_1,\ Exp)\ \mid \\
 & & Letrec\ Fun\ Case(nil,\ Exp,\ Var{:}Type{::}Var{:}List,\ Exp)\ \mid \\
 & & Letrec\ Fun\ Case((0, Var{:}Type),\ Exp,\ (Add_1, Var{:}Type),\ Exp)\ \mid \\
 & & Letrec\ Fun\ Case((nil, Var{:}Type),\ Exp, \\
 & & \qquad\qquad (Var{:}Type{::}Var{:}List, Var{:}Type),\ Exp) \\
Exp & = & OP_1(Exp)\ \mid \\
 & & OP_2(Exp, Exp)\ \mid \\
 & & if\ \ Exp\ \ then\ \ Exp\ \ else\ \ Exp\ \mid \\
 & & Var\ \mid \\
 & & Const \\
OP_1 & = & Fun \mid head \mid tail \\
OP_2 & = & Fun \mid cons \mid + \mid - \mid * \mid / \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid \ldots \\
Const & = & nil \mid 0 \mid 1 \mid \ldots \\
Add_1 & = & +(Var{:}Type, 1) \mid +(1, Var{:}Type) \\
Type & = & List \mid Type_b \\
Type_b & = & Int \mid Real \mid \ldots
\end{array}
$$

Le non-terminal Fun dénote l'ensemble des identificateurs de fonction et le non-terminal Var l'ensemble des variables (du premier ordre).

Dans le reste de ce document, nous utilisons les conventions de notation suivantes : P dénote une variable de la catégorie $Prog$; F une variable de la catégorie Def ; $e,\ e_1,\ e_2,\ e_3,\ e_4$ des variables de la catégorie Exp ; op une variable de la catégorie OP_1 ou OP_2 ; op_1 une variable de la catégorie OP_1 ; op_2 une variable de la catégorie OP_2 ; $f,\ g,\ h$ des variables de la catégorie Fun ; $k,\ k_1,\ k_2$ des variables de la catégorie $Const$; c une variable de la catégorie $Const$ différente de nil ; $x,\ x_1,\ x_2,\ x_3,\ l,\ l_1,\ l_2$ des variables de la catégorie Var ; $t,\ t_1,\ t_2,\ t_3$ des variables de la catégorie $Type$, et $t_b,\ t'_b$ des variables de la catégorie $Type_b$ (type de base).

Les propriétés que nous souhaitons pouvoir exprimer ici portant sur les entiers (en particulier, les longueurs de listes), le langage de propriétés L_{Prop} est une restriction de L_{Prog} aux fonctions de type $Int \to Int$ et $Int \times Int \to Int$. De ce fait, les informations de type deviennent redondantes et sont omises dans la syntaxe du langage L_{Prop}.

Langage de propriétés L_{Prop} :

$$
\begin{array}{lcl}
Prog & = & Def\ ;\ Prog\ \mid \\
 & & Def. \\
Def & = & Let\ Fun\ (Var,\ Exp)\ \mid \\
 & & Letrec\ Fun\ Case(0,\ Exp,\ Add_1,\ Exp)\ \mid \\
 & & Letrec\ Fun\ Case((0, Var),\ Exp,\ (Add_1, Var),\ Exp) \\
Exp & = & OP_1(Exp)\ \mid \\
 & & OP_2(Exp, Exp)\ \mid \\
 & & Var\ \mid \\
 & & Const \\
OP_1 & = & Fun \\
OP_2 & = & Fun \mid +\ \mid -\ \mid \ldots \\
Const & = & 0 \mid 1 \mid \ldots \\
Add_1 & = & +(Var, 1)\ \mid\ +(1, Var)
\end{array}
$$

3.1 L'interprétation abstraite

L'interprétation abstraite consiste à construire une sémantique abstraite dans laquelle on interprète les programmes. On peut exprimer dans la sémantique abstraite des propriétés qui représentent une approximation du comportement effectif du programme. Pour introduire les concepts de base de l'interprétation abstraite, nous considérons l'abstraction qui consiste à approximer les listes par leurs longueurs.

Pour définir une sémantique abstraite, il faut déterminer un domaine abstrait et établir sa correspondance avec le domaine standard (ou concret). Le tableau suivant présente les domaines abstraits associés aux différents domaines standards qui nous intéressent (c'est-à-dire ceux utilisés dans notre langage de programmation L_{Prog}).

Type	Domaine standard	Domaine abstrait
List	$List\_$	$\mathbb{N}_-^{\top} = \mathbb{N}_- \cup \{\top\}$
Int	$\mathbb{N}_-$	$\{-\}$
Real	$\mathbb{R}_-$	$\{-\}$

Pour l'abstraction proposée ici (c'est-à-dire l'approximation des listes par leurs longueurs), nous avons choisi d'abstraire tout domaine autre que celui dénoté par le type *List* par un domaine à un seul élément $-$ (*indéfini*). Le domaine abstrait $\mathbb{N}_-^{\top}$ associé au type *List* est le treillis $\mathbb{N}_-^{\top}$ engendré par l'ordre partiel $\sqsubseteq$ suivant :

- $\forall x \in \mathbb{N}_-^{\top}\ .\ x \sqsubseteq \top$,
- $\forall x \in \mathbb{N}\ .\ - \sqsubseteq x$.

Nous caractérisons maintenant les relations entre les différents domaines standards et abstraits par le biais de fonctions d'abstraction et de concrétisation. Pour servir de base à une interprétation abstraite, ces deux fonctions doivent former une connection de Galois. En fait, on ne considère ici que le type *List* et $\mathbb{N}_-^{\top}$ car les opérations sont triviales pour tous les autres domaines (par exemple, $\forall x \in \mathbb{N}_-\ .\ abs(x) = -$ et $Conc(-) = \mathbb{N}_-$).

On définit le plus petit majorant $\sqcup$ sur $\mathbb{N}_{-}^{\top}$ par :

$\forall a \in \mathbb{N}_{-}^{\top} \,.\, \forall b \in \mathbb{N}_{-}^{\top} \,.\, a \sqcup b =$ (le plus petit $x \in \mathbb{N}_{-}^{\top} \,.\, (a \sqsubseteq x \wedge b \sqsubseteq x)$).

La fonction d'abstraction pour le type *List* est la suivante :

$$\begin{array}{lll} Abs \;:\; \mathcal{P}(List_{-}) & \rightarrow & \mathbb{N}_{-}^{\top} \\ \quad Abs(E) & = & \sqcup\{abs(x) \mid x \in E\} \end{array}$$

avec

$$\begin{array}{lll} abs \;:\; List_{-} \rightarrow \mathbb{N}_{-}^{\top} & & \\ \quad abs(-) & = & - \\ \quad abs(nil) & = & 0 \\ \quad abs(x_1 :: \ldots :: x_n :: nil) & = & n \end{array}$$

La fonction de concrétisation pour le type *List* est la suivante :

$$\begin{array}{lll} Conc \;:\; \mathbb{N}_{-}^{\top} \rightarrow \mathcal{P}(List_{-}) & & \\ \quad Conc(-) & = & \{-\} \\ \quad Conc(\top) & = & List_{-} \\ \quad Conc(n) & = & \{l \in List_{-} \mid length(l) = n\} \end{array}$$

Il est facile de montrer que $Abs \circ Conc = id_{\mathbb{N}_{\perp}^{\top}}$, et que $Conc \circ Abs \supseteq id_{List_{\perp}}$, ce qui assure que nous avons effectivement construit une connection de Galois [CC77, CC92].

Les domaines standards et abstraits étant définis, nous pouvons maintenant présenter les sémantiques standard et abstraite, avant de préciser la relation qu'elles doivent satisfaire.

Définition de la fonction sémantique S (sémantique standard) :

La fonction sémantique S prend en entrée un environnement ? et une phrase du langage L_{Prog}. L'environnement ? associe une valeur du domaine standard à tout identificateur de variable libre (fonction ou variable du premier ordre). On utilise $\overline{c}$ pour distinguer la valeur sémantique de c de sa valeur syntaxique et $<>$ pour représenter la valeur *liste vide*. Y est l'opérateur de point fixe.

$S(?, F; P) = S(?\,[S(?, F)/f], P)$
$\quad$ *si* $F = Let\ f \ldots$ *ou* $F = Letrec\ f \ldots$
$S(?, Let\ f\ (x{:}t,\ e)) = \lambda v.\ S(?\,[v/x], e)$
$S(?, Letrec\ f\ Case(0,\ e_1,\ +(x{:}t, 1),\ e_2)) =$
$\quad \mathbf{Y}(\lambda g.\ \lambda v.\ S(Case_{Int})(v, S(?\,[g/f], e_1), S(?\,[g/f,\ v-1/x], e_2))$
$S(?, Letrec\ f\ Case(0,\ e_1,\ +(1, x{:}t),\ e_2)) =$
$\quad \mathbf{Y}(\lambda g.\ \lambda v.\ S(Case_{Int})(v, S(?\,[g/f], e_1), S(?\,[g/f,\ v-1/x], e_2))$
$S(?, Letrec\ f\ Case(nil,\ e_1,\ x{:}t{::}l{:}List,\ e_2)) =$
$\quad \mathbf{Y}(\lambda g.\ \lambda v.\ S(Case_{List})(v, S(?\,[g/f], e_1),$
$\qquad\qquad S(?\,[g/f,\ S(head)(v)/x,\ S(tail)(v)/l], e_2))$
$S(?, Letrec\ f\ Case((0, x_1{:}t_1),\ e_1,\ (+(x{:}t, 1), x_2{:}t_2),\ e_2)) =$
$\quad \mathbf{Y}(\lambda g.\ \lambda(v, w).\ S(Case_{Int})(v, S(?\,[g/f,\ w/x_1], e_1), S(?\,[g/f,\ v-1/x,\ w/x_2], e_2))$
$S(?, Letrec\ f\ Case((0, x_1{:}t_1),\ e_1,\ (+(1, x{:}t), x_2{:}t_2),\ e_2)) =$
$\quad \mathbf{Y}(\lambda g.\ \lambda(v, w).\ S(Case_{Int})(v, S(?\,[g/f,\ w/x_1], e_1), S(?\,[g/f,\ v-1/x,\ w/x_2], e_2))$
$S(?, Letrec\ f\ Case((nil, x_1{:}t_1),\ e_1,\ x{:}t{::}l{:}List, x_2{:}t_2,\ e_2)) =$
$\quad \mathbf{Y}(\lambda g.\ \lambda(v, w).\ S(Case_{List})(v, S(?\,[g/f,\ w/x_1], e_1),$
$\qquad\qquad S(?\,[g/f,\ S(head)(v)/x,\ S(tail)(v)/l,\ w/x_2], e_2))$
$S(?, op_1(e)) = S(op_1)(S(?, e))$ avec $op_1 \notin Fun$
$S(?, f(e)) = ?\,(f)(S(?, e))$
$S(?, op_2(e_1, e_2)) = S(op_2)(S(?, e_1), S(?, e_2))$ avec $op_2 \notin Fun$
$S(?, f(e_1, e_2)) = ?\,(f)(S(?, e_1), S(?, e_2))$
$S(?, if\ e\ then\ e_1\ else\ e_2) = S(Cond)(S(?, e), S(?, e_1), S(?, e_2))$
$S(?, x) = ?\,(x)$
$S(?, c) = S(c)$
$S(?, nil) = S(nil)$
$S(Cond)(v, v_1, v_2) = \overline{si}\ v\ \overline{alors}\ v_1\ \overline{sinon}\ v_2$
$S(head)(x{::}l) = x$
$S(tail)(x{::}l) = l$
$S(cons)(x, l) = x{::}l$
$S(+)(x_1, x_2) = \overline{x_1 + x_2}$
$S(-)(x_1, x_2) = \overline{x_1 - x_2}$
$S(*)(x_1, x_2) = \overline{x_1 * x_2}$
$S(/)(x_1, x_2) = \overline{x_1/x_2}$
$S(Case_{Int})(v, v_1, v_2) = \overline{si}\ v = \overline{0}\ \overline{alors}\ v_1\ \overline{sinon}\ v_2$
$S(Case_{List})(v, v_1, v_2) = \overline{si}\ v =<>\ \overline{alors}\ v_1\ \overline{sinon}\ v_2$
$S(c) = \overline{c}$
$S(nil) =<>$

Définition de la fonction sémantique S_a (sémantique abstraite) :

La fonction sémantique S_a prend en entrée un environnement abstrait $?_a$ et une phrase du langage L_{Prog}. L'environnement abstrait $?_a$ associe une valeur du domaine abstrait à tout identificateur de variable libre. La fonction S_a est définie exactement

comme S, sauf pour les opérateurs de base :

$$\begin{array}{lcll}
S_a(Cond)(v, v_1, v_2) & = & v_1 \sqcup v_2 & \\
S_a(head)(x) & = & - & \\
S_a(tail)(x) & = & \overline{x - 1} & \\
S_a(cons)(x_1, x_2) & = & \overline{x_2 + 1} & \\
S_a(+)(x_1, x_2) & = & - & \\
S_a(-)(x_1, x_2) & = & - & \\
S_a(*)(x_1, x_2) & = & - & \\
S_a(/)(x_1, x_2) & = & - & \\
S_a(Case_{Int})(v, v_1, v_2) & = & v_1 \sqcup v_2 & \\
S_a(Case_{List})(v, v_1, v_2) & = & \overline{si}\ v = \overline{0}\ \overline{alors}\ v_1\ \overline{sinon}\ v_2 & \\
S_a(c) & = & -_t & \text{avec } c : t \\
S_a(nil) & = & \overline{0} &
\end{array}$$

Pour que l'analyse par interprétation abstraite apporte une information formelle sur un programme, il faut que les sémantiques standard et abstraite soient liées par une relation de correspondance définie précisément. Nous sommes à présent en mesure de présenter la relation de correction de l'interprétation abstraite :

Définition 11
La sémantique (ou interprétation) abstraite S_a est dite correcte par rapport à la sémantique standard S selon la relation de correspondance R si et seulement si :

$$\forall e\ .\ (\forall x \in ?\ .\ R(?\,(x), ?_a(x))) \Rightarrow R(S(?\,, e), S_a(?_a, e))$$

avec :

$$\begin{array}{lcll}
R(l, n) & \equiv & (abs(l) = n) & si\ l \in List \\
R(f, f_a) & \equiv & (\forall x\ .\ \forall x_a\ .\ R(x, x_a) \Rightarrow R(f(x), f_a(x_a))) & si\ f \in D \to D \\
R(f, f_a) & \equiv & (\forall x\ .\ \forall y\ .\ \forall x_a\ .\ \forall y_a\ .\ R(x, x_a) \wedge R(y, y_a) \Rightarrow R(f(x, y), f_a(x_a, y_a))) & \\
 & & & si\ f \in D \times D \to D
\end{array}$$

Il a été établi [CC77] que la condition suivante est suffisante pour assurer la correction de l'interprétation abstraite :

Propriété 25
Si pour toute constante k et tout opérateur de base n-aire du langage op,

$$S(k) \in Conc(S_a(k))$$

et

$$\forall v_1, \ldots, v_n\ .\ S(op)(v_1, \ldots, v_n) \in Conc(S_a(op)(abs(v_1), \ldots, abs(v_n)))$$

alors l'interprétation abstraite S_a est correcte au sens de la Définition 11.

Pour montrer la correction de l'interprétation abstraite définie ici, il suffit donc de vérifier cette condition pour chaque constante et opérateur de base du langage L_{Prog}. Nous considérons uniquement ici les valeurs de base de type Int, les réels se traitant de la même manière.

- $S(Cond)(v, v_1, v_2) \in Conc(S_a(Cond)(abs(v), abs(v_1), abs(v_2)))$ puisque
  $Conc(S_a(Cond)(abs(v), abs(v_1), abs(v_2))) = Conc(abs(v_1) \sqcup abs(v_2))$ et,
  si $v = vrai$ alors $S(Cond)(v, v_1, v_2) = v_1$,
  avec $v_1 \in Conc(abs(v_1)) \subseteq Conc(abs(v_1) \sqcup abs(v_2))$.
  On applique le même raisonnement au cas $v = faux$.
- $S(head)(x::l) \in Conc(-_{Int})$ si x est de type Int puisque
  $S(head)(x::l) = x$ et $Conc(-_{Int}) = \mathbb{N}_-$.
- $S(tail)(x::l) \in Conc(abs(x::l) - 1)$ puisque
  $abs(x::l) = length(x::l) = length(l) + 1$ et $l \in Conc(length(l))$.
- $S(cons)(v_1, v_2) \in Conc(1 + abs(v_2))$ car
  $abs(v_2) = length(v_2)$ et $Conc(1+abs(v_2)) = \{l \in List_- \mid length(l) = 1+abs(v_2)\}$,
  et on a bien $length(S(cons)(v_1, v_2)) = 1 + length(v_2)$.
- $S(+)(v_1, v_2) \in Conc(-_{Int}) = \mathbb{N}_-$.
- $S(-)(v_1, v_2) \in Conc(-_{Int})$, $S(*)(v_1, v_2) \in Conc(-_{Int})$ et $S(/)(v_1, v_2) \in Conc(-_{Int})$
  pour les mêmes raisons que $S(+)(v_1, v_2) \in Conc(-_{Int})$.
- $S(Case_{Int})(v, v_1, v_2) \in Conc(abs(v_1) \sqcup abs(v_2))$ pour la même raison que précédemment.
- $S(Case_{List})(v, v_1, v_2) \in Conc(S_a(Case_{List})(abs(v), abs(v_1), abs(v_2))$ car
  si $v =<>$ alors $abs(v) = \overline{0}$
  et on a bien $v_1 \in Conc(abs(v_1))$ puisque $Conc \circ Abs \supseteq id_{List_\perp}$;
  et si $v \neq <>$ alors $abs(v) \neq \overline{0}$
  et on conclut de la même façon.
- $S(c) \in Conc(-_{Int})$ si c est de type Int puisque $Conc(-_{Int}) = \mathbb{N}_-$.
- $S(nil) \in Conc(\overline{0})$ puisque $Conc(\overline{0}) = \{l \in List_- \mid length(l) = 0\}$.

3.2 Expression de l'analyse par transformation de programmes

L'abstraction *longueur* consiste à remplacer toute information de type liste par une information de type *longueur de liste*. En particulier, la constante *nil* est abstraite par l'entier 0 et l'opérateur prédéfini *cons* par la fonction $\lambda x.\ (x + 1)$. Le domaine abstrait est donc celui des entiers naturels. Notre langage de propriétés L_{Prop} permettant d'exprimer des fonctions sur les entiers, on peut en fait réaliser l'interprétation abstraite comme une transformation de programmes de L_{Prog} vers L_{Prop}. C'est de cette manière que l'analyse décrite plus haut est effectivement implantée dans notre prototype (cf. Annexe D). Cette transformation se réduit à un remplacement des opérateurs de base sur les listes (comme *cons*) par des opérations de base sur les entiers (comme +1).

Puisque les arguments de type $t \neq List$ sont systématiquement abstraits en la valeur $-$ et n'apportent aucune information dans l'analyse, nous choisissons de les faire disparaître dans la transformation. Ainsi une fonction de type $List \times Int \rightarrow List$ par exemple est transformée en une fonction de type $Int \rightarrow Int$. De la même façon, nous ne décrivons pas le traitement des fonctions définies par récurrence sur les entiers puisqu'elles sont abstraites en la fonction indéfinie.

Les règles de transformation sont les suivantes :

$$
\begin{array}{l}
T_a(F;P) = T_a(F);T_a(P) \\
T_a(Let\ f\ (x{:}List,\ e)) = Let\ f\ (x,\ T_a(e)) \\
T_a(Letrec\ f\ Case(nil,\ e_1,\ x{:}t_b{::}l{:}List,\ e_2)) \\
\qquad = Letrec\ f\ Case(0,\ T_a(e_1),\ +(l,1),\ T_a(e_2)) \\
T_a(Letrec\ f\ Case((nil, x{:}t'_b),\ e_1,\ (x_1{:}t_b{::}l{:}List, x_2{:}t'_b),\ e_2) \\
\qquad = Letrec\ f\ Case(0,\ T_a(e_1),\ +(l,1),\ T_a(e_2)) \\
T_a(Letrec\ f\ Case((nil, l{:}List),\ e_1,\ (x_1{:}t_b{::}l_1{:}List, l_2{:}List),\ e_2)) \\
\qquad = Letrec\ f\ Case((0,l),\ T_a(e_1),\ (+(l_1,1), l_2),\ T_a(e_2)) \\
T_a(f_{List\rightarrow List}(e)) = f(T_a(e)) \\
T_a(tail(l)) = -(T_a(l),1) \\
T_a(f_{List\times List\rightarrow List}(e_1,e_2)) = f(T_a(e_1),T_a(e_2)) \\
T_a(f_{List\times t_b\rightarrow List}(e_1,e_2)) = f(T_a(e_1)) \\
T_a(f_{t_b\times List\rightarrow List}(e_1,e_2)) = f(T_a(e_2)) \\
T_a(cons(x,l)) = +(T_a(l),1) \\
T_a(x_{List}) = x \\
T_a(nil) = 0
\end{array}
$$

On remarque que la transformation nécessite de connaître le type des fonctions apparaissant dans les expressions. Plus précisément, il est nécessaire de pouvoir distinguer un type $List$ d'un type de base. Nous proposons en Annexe A un système d'inférence de type fournissant les informations nécessaires à l'abstraction.

La Propriété 26 établit la correspondance entre la transformation T_a et la définition de l'interprétation abstraite S_a.

Propriété 26

$$S_a(?_a, P) \equiv S(?_a, T_a(P))$$

avec

$$
\begin{array}{lcll}
x \equiv x' & \Longleftrightarrow & x = x' & si\ x \in I\!N \\
f \equiv f' & \Longleftrightarrow & f = f' & si\ f : I\!N \rightarrow I\!N \\
 & & f = f' & si\ f : I\!N \times I\!N \rightarrow I\!N \\
 & & f' = \lambda x.\ f(x,-) & si\ f : I\!N \times \{-\} \rightarrow I\!N \\
 & & f' = \lambda x.\ f(-,x) & si\ f : \{-\} \times I\!N \rightarrow I\!N
\end{array}
$$

Pour illustrer cette transformation, nous reprenons l'exemple du programme *Reverse* utilisé dans le Chapitre 1.

Nous l'écrivons dans la syntaxe du langage de programmation L_{Prog} :

$$\begin{array}{l} \mathit{Letrec\ Append\ Case}((\mathit{nil}, m{:}\mathit{Int}),\ \mathit{cons}(m, \mathit{nil}),\\ \qquad\qquad (n{:}\mathit{Int}{::}l{:}\mathit{List}, m{:}\mathit{Int}),\ \mathit{cons}(n, \mathit{Append}(l, m)));\\ \mathit{Letrec\ Reverse\ Case}(\mathit{nil},\ \mathit{nil},\ n{:}\mathit{Int}{::}l{:}\mathit{List},\ \mathit{Append}(\mathit{Reverse}(l), n)). \end{array}$$

Le système d'inférence de type (cf. Annexe A) permet d'associer à *Append* le type $\mathit{List} \times \mathit{Int} \to \mathit{List}$ et à *Reverse* le type $\mathit{List} \to \mathit{List}$.

Nous détaillons successivement la transformation de *Append* puis de *Reverse* :

$$\begin{array}{l} T_a(\mathit{Letrec\ Append\ Case}((\mathit{nil}, m{:}\mathit{Int}),\ \mathit{cons}(m, \mathit{nil}),\\ \qquad\qquad (n{:}\mathit{Int}{::}l{:}\mathit{List}, m{:}\mathit{Int}),\ \mathit{cons}(n, \mathit{Append}(l, m)))) =\\ \mathit{Letrec\ Append\ Case}(0,\ T_a(\mathit{cons}(m, \mathit{nil})),\ +(l, 1),\ T_a(\mathit{cons}(n, \mathit{Append}(l, m)))) =\\ \mathit{Letrec\ Append\ Case}(0,\ +(T_a(\mathit{nil}), 1),\ +(l, 1),\ +(T_a(\mathit{Append}(l, m)), 1)) =\\ \mathit{Letrec\ Append\ Case}(0,\ +(0, 1),\ +(l, 1),\ +(\mathit{Append}(T_a(l)), 1)) =\\ \mathit{Letrec\ Append\ Case}(0,\ +(0, 1),\ +(l, 1),\ +(\mathit{Append}(l), 1)) \end{array}$$

$$\begin{array}{l} T_a(\mathit{Letrec\ Reverse\ Case}(\mathit{nil},\ \mathit{nil},\ n{:}\mathit{Int}{::}l{:}\mathit{List},\ \mathit{Append}(\mathit{Reverse}(l), n))) =\\ \mathit{Letrec\ Reverse\ Case}(0,\ T_a(\mathit{nil}),\ +(l, 1),\ T_a(\mathit{Append}(\mathit{Reverse}(l), n))) =\\ \mathit{Letrec\ Reverse\ Case}(0,\ 0,\ +(l, 1),\ \mathit{Append}(T_a(\mathit{Reverse}(l)))) =\\ \mathit{Letrec\ Reverse\ Case}(0,\ 0,\ +(l, 1),\ \mathit{Append}(\mathit{Reverse}(T_a(l)))) =\\ \mathit{Letrec\ Reverse\ Case}(0,\ 0,\ +(l, 1),\ \mathit{Append}(\mathit{Reverse}(l))) \end{array}$$

La transformation T_a produit donc le programme suivant (dans lequel nous avons renommé les variables pour une plus grande clarté) :

$$\begin{array}{l} \mathit{Letrec\ Lappend\ Case}(0,\ +(0, 1),\ +(n, 1),\ +(\mathit{Lappend}(n), 1));\\ \mathit{Letrec\ Lreverse\ Case}(0,\ 0,\ +(n, 1),\ \mathit{Lappend}(\mathit{Lreverse}(n))). \end{array}$$

L'abstraction de *Append* (fonction binaire, de type $\mathit{List} \times \mathit{Int} \to \mathit{List}$) en *Lappend* (fonction unaire, de type $\mathit{Int} \to \mathit{Int}$) illustre l'élagage réalisé par T_a sur les arguments dont le type est différent du type *List* : ici, le second argument de *Append* (de type *Int*) a été supprimé.

On peut vérifier que ces programmes sont identiques aux fonctions *Lappend* et *Lreverse* du Chapitre 1.

Chapitre 4

Inférence de schéma

Nous avons montré dans le chapitre précédent comment l'abstraction d'un programme de L_{Prog} pouvait s'exprimer comme une transformation de programme de L_{Prog} vers L_{Prop}, qui est lui-même un sous-ensemble de L_{Prog}. La propriété que nous cherchons à vérifier est aussi exprimée dans L_{Prop}. L'étape suivante de la méthode consiste à comparer ces deux fonctions. Pour ce faire, il faut trouver le plus petit schéma de fonction qui les contient toutes les deux. Nous devons donc associer à chacune d'elles un schéma de fonction. Cette phase est réalisée par le système d'inférence de schéma que nous présentons dans ce chapitre. La reconnaissance d'un schéma se fait uniquement sur des critères syntaxiques. Le programme abstrait et la propriété doivent donc être écrits selon des règles très strictes, sous peine de ne pas être reconnus comme appartenant à l'un des schémas d'intérêt. Cela peut être gênant en ce qui concerne la propriété, car elle est directement écrite par l'utilisateur. De même, le programme abstrait est issu d'une transformation du programme initial; il est donc d'une forme similaire. Par exemple, nous avons vu dans la Partie 3.2 que le résultat de l'abstraction de la fonction *Append* est la fonction *Lappend*, qui s'écrit dans notre langage :

$$Letrec\ Lappend\ Case(0,\ +(0,1),\ +(n,1),\ +(Lappend(n),1))$$

Nous souhaitons pouvoir associer le schéma S_2^1 à cette fonction. Nous rappelons que S_2^1 est défini par :

$$\begin{aligned} S_2^1 \quad = \quad & \{f \mid f(0) = k \\ & \qquad f(n+1) = g(f(n))\ ,\ k \in \mathbb{N},\ g \in S_1^1\} \end{aligned}$$

Ce qui s'écrit dans la syntaxe du langage L_{Prog} :

$$S_2^1 \quad = \quad \{f \mid Letrec\ f\ Case(0,\ k,\ +(n,1),\ g(f(n)))\ ,\ k \in \mathbb{N},\ g \in S_1^1\}$$

On se rend compte que la fonction *Lappend* telle qu'elle est exprimée ci-dessus n'est pas une instance directe du schéma S_2^1. En particulier, $+(0,1)$ ne correspond pas directement à k (constante), et la fonction binaire $+$ n'est pas une instance de la fonction unaire g du schéma. Deux options sont possibles pour résoudre cette difficulté :

1. Définir une phase préalable de normalisation.

2. Proposer un système d'inférence de schéma plus sophistiqué capable de tenir compte de ces variations.

Nous choisissons ici une solution mixte. D'une part, nous définissons une procédure de normalisation pour effectuer toutes les simplifications de nature arithmétique (comme remplacer $+(0,1)$ par 1). D'autre part, nous définissons des versions unaires des fonctions binaires de base, lorsque celles-ci sont appelées avec un argument constant (comme la fonction $succ = \lambda x.\ (x+1)$). Ces fonctions prédéfinies sont utilisables par le système d'inférence de schéma, qui prend en entrée un programme abstrait normalisé ou une propriété normalisée. Nous détaillons la procédure de normalisation dans la Partie 4.1, avant de présenter l'inférence de schéma proprement dite dans la Partie 4.2.

4.1 Normalisation

La normalisation procède en réordonnant les expressions et en effectuant toutes les opérations arithmétiques faisant intervenir des constantes. Elle ne s'applique que sur les expressions (dénotées dans la grammaire par Exp).

On utilise la notation $\overline{a+b}$ pour dénoter le résultat de l'évaluation de $a+b$. Les règles de réécriture pour la normalisation sont les suivantes :

$$
\begin{array}{lcll}
+(e,0) & \to & e & \\
+(k_1,k_2) & \to & \overline{k_1+k_2} & \\
-(e,0) & \to & e & \\
-(k_1,k_2) & \to & \overline{k_1-k_2} & \\
+(k,e) & \to & +(e,k) & \text{avec } e \notin Const \\
+(+(e_1,e_2),e_3) & \to & +(e_1,+(e_2,e_3)) & \\
-(-(e_1,e_2),e_3) & \to & -(e_1,+(e_2,e_3)) & \\
-(+(e_1,e_2),e_3) & \to & +(e_1,-(e_2,e_3)) &
\end{array}
$$

Les règles sont appliquées dans l'ordre où elles sont définies. La stratégie utilisée consiste à déplacer vers la droite les constantes pour effectuer toutes les simplifications possibles. On peut associer un ordre strictement décroissant à ce système de réécriture (cf. Annexe B) : il est donc convergent.

Reprenons l'exemple du programme *Reverse*. Sa version abstraite (présentée dans la Partie 3.2) est :

$$
\begin{array}{l}
Letrec\ Lappend\ Case(0,\ +(0,1),\ +(n,1),\ +(Lappend(n),1)); \\
Letrec\ Lreverse\ Case(0,\ 0,\ +(n,1),\ Lappend(Lreverse(n))).
\end{array}
$$

Après application de la normalisation, on obtient le programme :

$$
\begin{array}{l}
Letrec\ Lappend\ Case(0,\ 1,\ +(n,1),\ +(Lappend(n),1)); \\
Letrec\ Lreverse\ Case(0,\ 0,\ +(n,1),\ Lappend(Lreverse(n))).
\end{array}
$$

4.2 Système d'inférence de schéma

L'inférence de schéma permet de reconnaître les différents schémas unaires ou binaires présentés dans les Parties 2.2.2, 2.2.3 et 2.3.1.

Elle dispose des définitions de base suivantes :

$$\begin{array}{lcl} id & = & \lambda x.\ x \\ \forall k,\ \overline{k} & = & \lambda x.\ k \\ \forall k,\ plusk & = & \lambda x.\ (x + k) \end{array}$$

et peut aussi construire une application de la fonction identité ou d'une fonction constante à partir d'une variable ou d'une constante (par exemple, remplacer k par $\overline{k}(x)$).

Par la suite, on utilise la notation s pour représenter un schéma parmi ceux étudiés dans le Chapitre 2, c'est-à-dire S^1_n, S^2_n, P_n, I_n, I'_n, I''_n, $B^1(X_1, X_2)$, $B^2(X_1, X_2)$ et $B^3(X)$. On utilise ? pour dénoter un contexte (ou environnement). Un contexte est un ensemble de paires (*identificateur*,*schéma*) associant un identificateur de fonction avec le schéma auquel cette fonction appartient. $?[f : s]$ représente le contexte obtenu en ajoutant la paire $(f,\ s)$ à ?. $? \vdash e : s$ signifie que l'expression e est de schéma s dans le contexte ?. Une règle d'inférence est composée de deux parties séparées d'une barre horizontale : la partie supérieure représente les prémisses (ou hypothèses) et la partie inférieure la conclusion.

Nous présentons un système d'inférence pour les schémas unaires dans la Partie 4.2.1 et un système d'inférence pour les schémas binaires dans la Partie 4.2.2.

4.2.1 Système d'inférence de schémas unaires

Les axiomes et règles d'inférence de schémas unaires sont les suivants :

(P_1)
$$\frac{? \vdash Let\ f\ (x,\ e) : s_1 \quad ?[f : s_1] \vdash P : s_2}{? \vdash Let\ f\ (x,\ e); P : s_2}$$

(P_2)
$$\frac{? \vdash Letrec\ f\ Case(0,\ e_1,\ +(x,1),\ e_2) : s_1 \quad ?[f : s_1] \vdash P : s_2}{? \vdash Letrec\ f\ Case(0,\ e_1,\ +(x,1),\ e_2); P : s_2}$$

(Prim_1)
$$? \vdash Let\ f\ (x,\ x) : S^1_1$$

(Prim_2)
$$? \vdash Let\ f\ (x,\ k) : S^1_1$$

(Prim_3)
$$? \vdash Let\ f\ (x,\ +(x,k)) : S^1_1$$

(Base$_1$)
$$? \vdash id : S_1^1$$

(Base$_2$)
$$? \vdash \overline{k} : S_1^1$$

(Base$_3$)
$$? \vdash plusk : S_1^1$$

(Ssun)
$$\frac{? \vdash g : S_i^1}{? \vdash Letrec\ f\ Case(0,\ k,\ +(x,1),\ g(f(x))) : S_{i+1}^1}$$

(Scun)
$$\frac{? \vdash g : S_{i_g}^1 \quad ? \vdash h : S_{i_h}^1}{? \vdash Letrec\ f\ Case(0,\ k,\ +(x,1),\ g(f(h(x)))) : S_i^2} \quad i = max(i_g, i_h)$$

(Fonc)
$$?[f : s] \vdash f : s$$

Le coeur du système d'inférence est constitué des règles (Prim$_1$), (Prim$_2$), (Prim$_3$), (Ssun) et (Scun). Les trois premières reposent sur la définition de S_1^1 (cf. Définition 7) ; (Ssun) et (Scun) correspondent aux définitions de S_{i+1}^1 et S_i^2 respectivement (cf. Définitions 7 et 8). Les règles (P$_1$) et (P$_2$) servent à propager l'inférence des schémas de fonctions au programme entier.

On peut remarquer que ce système conduit directement à un algorithme d'inférence car on peut lui associer un ordre strict dans le calcul des schémas, et il n'y a aucune ambiguïté dans le choix d'une règle à appliquer. Pour obtenir l'algorithme, il suffit de lire les règles de bas en haut (conclusions vers hypothèses) et de gauche à droite. Par exemple, la règle (P$_1$) se lit : «Pour prouver que l'expression $Let\ f\ (x,\ e); P$ est de schéma s_2 dans un contexte ?, il suffit de montrer que l'expression $Let\ f\ (x,\ e)$ est de schéma s_1 dans le contexte ?, puis que l'expression P est de schéma s_2 dans le contexte ? augmenté de $[f : s_1]$.». On remarque que s_1 est connu grâce à la preuve de la première hypothèse avant d'être utilisé dans la preuve de la seconde.

Pour illustrer le fonctionnement du système d'inférence de schéma, nous poursuivons l'étude de l'exemple du programme *Reverse*. Le programme *Reverse* abstrait normalisé (cf. Partie 4.1) est :

$Letrec\ Lappend\ Case(0,\ 1,\ +(n,1),\ +(Lappend(n),1));$
$Letrec\ Lreverse\ Case(0,\ 0,\ +(n,1),\ Lappend(Lreverse(n))).$

Après utilisation[1] des définitions de base, on obtient le programme suivant :
$Letrec\ Lappend\ Case(0,\ 1,\ +(n,1),\ plus1(Lappend(n)));$

1. En pratique, l'utilisation des définitions de base est intimement liée au processus d'inférence de schéma proprement dit.

$Letrec\ Lreverse\ Case(0,\ 0,\ +(n,1),\ Lappend(Lreverse(n)))$.

Ici, on a appliqué la transformation : $+(e,k) \rightarrow plusk(e)$.

Nous détaillons à présent l'application de l'inférence de schéma sur ce programme. Pour plus de clarté, nous définissons :

$$\begin{array}{lcl} Prog_1 & = & Letrec\ Lappend\ Case(0,\ 1,\ +(n,1),\ plus1(Lappend(n)));\\ & & Prog_2\\ Prog_2 & = & Letrec\ Lreverse\ Case(0,\ 0,\ +(n,1),\ Lappend(Lreverse(n))). \end{array}$$

Au départ, le système d'inférence de schéma prend en entrée un contexte vide et le programme entier. Il doit produire en résultat le schéma s auquel appartient le programme. On cherche donc à prouver :

$$\vdash Prog_1 : s$$

c'est-à-dire

$$\vdash Letrec\ Lappend\ Case(0,\ 1,\ +(n,1),\ plus1(Lappend(n)));Prog_2 : s$$

La seule règle qui s'applique est la règle (P_2) :

$$\frac{\begin{array}{ll} & \vdash\ Letrec\ Lappend\ Case(0,\ 1,\ +(n,1),\ plus1(Lappend(n))) : s_1\\ [Lappend : s_1] & \vdash\ Prog_2 : s \end{array}}{\vdash Letrec\ Lappend\ Case(0,\ 1,\ +(n,1),\ plus1(Lappend(n)));Prog_2 : s}$$

• La prochaine étape est la résolution de :

$$\vdash Letrec\ Lappend\ Case(0,\ 1,\ +(n,1),\ plus1(Lappend(n))) : s_1$$

La seule règle qui s'applique est la règle (Ssun) :

$$\frac{\vdash plus1 : S_i^1}{\vdash Letrec\ Lappend\ Case(0,\ 1,\ +(n,1),\ plus1(Lappend(n))) : S_{i+1}^1}$$

Il reste alors à prouver l'hypothèse : $\vdash plus1 : S_i^1$, ce que le système établit par application de l'axiome (Base_3) :

$$\vdash plus1 : S_1^1$$

On obtient donc :

$$\vdash Letrec\ Lappend\ Case(0,\ 1,\ +(n,1),\ plus1(Lappend(n))) : S_2^1$$

• L'environnement est maintenant augmenté de l'hypothèse $\{Lappend : S_2^1\}$ et il reste à typer $Prog_2$, c'est-à-dire trouver s tel que :

$$\{Lappend : S_2^1\} \vdash Letrec\ Lreverse\ Case(0,\ 0,\ +(n,1),\ Lappend(Lreverse(n))) : s$$

La seule règle qui s'applique est la règle (Ssun) :

$$\frac{\{Lappend : S_2^1\} \vdash Lappend : S_i^1}{\{Lappend : S_2^1\} \vdash Letrec\ Lreverse\ Case(0,\ 0,\ +(n,1),\ Lappend(Lreverse(n))) : S_{i+1}^1}$$

Il reste à montrer l'hypothèse :

$$\{Lappend : S_2^1\} \vdash Lappend : S_i^1$$

Le système applique alors l'axiome (Fonc) :

$$\{Lappend : S_2^1\} \vdash Lappend : S_2^1$$

On obtient donc :

$$\{Lappend : S_2^1\} \vdash Letrec\ Lreverse\ Case(0,\ 0,\ +(n,1),\ Lappend(Lreverse(n))) : S_3^1$$

C'est-à-dire :

$$\{Lappend : S_2^1\} \vdash Prog_2 : S_3^1$$

On en conclut donc :

$$\vdash Prog_1 : S_3^1$$

Ainsi, ce système d'inférence de schéma rend le schéma S_3^1 pour le programme *Lreverse*.

4.2.2 Système d'inférence de schémas binaires

Les axiomes et règles d'inférence de schémas binaires sont les suivants :

(P_1)

$$\frac{? \vdash Let\ f\ (x,\ e) : s_1 \quad ?\,[f : s_1] \vdash P : s_2}{? \vdash Let\ f\ (x,\ e); P : s_2}$$

(P_2)

$$\frac{? \vdash Letrec\ f\ Case(0,\ e_1,\ +(x,1),\ e_2) : s_1 \quad ?\,[f : s_1] \vdash P : s_2}{? \vdash Letrec\ f\ Case(0,\ e_1,\ +(x,1),\ e_2); P : s_2}$$

(P_3)

$$\frac{? \vdash Letrec\ f\ Case((0,x_1),\ e_1,\ (+(x_2,1),x_3),\ e_2) : s_1 \quad ?\,[f : s_1] \vdash P : s_2}{? \vdash Letrec\ f\ Case((0,x_1),\ e_1,\ (+(x_2,1),x_3),\ e_2); P : s_2}$$

($Prim_1$)

$$? \vdash Let\ f\ (x,\ k) : P_1$$

($Prim_2$)

$$? \vdash Let\ f\ (x,\ +(x,k)) : I_1$$

(Prim$_2$')

$$\frac{k > 0}{? \vdash Let\ f\ (x,\ \ +(x,k)) : I'_1}$$

(Prim$_2$")

$$\frac{k > 1}{? \vdash Let\ f\ (x,\ \ +(x,k)) : I''_1}$$

(Base$_1$)

$$? \vdash \overline{k} : P_1$$

(Base$_2$)

$$? \vdash id : I_1$$

(Base$_3$)

$$? \vdash plusk : I_1$$

(Base$_3$')

$$\frac{k > 0}{? \vdash plusk : I'_1}$$

(Base$_3$")

$$\frac{k > 1}{? \vdash plusk : I''_1}$$

(Ssun-P)

$$\frac{? \vdash g : P_i}{? \vdash Letrec\ f\ Case(0,\ k,\ \ +(x,1),\ g(f(x))) : P_{i+1}}$$

(Ssun-I)

$$\frac{? \vdash g : I'_i}{? \vdash Letrec\ f\ Case(0,\ k,\ \ +(x,1),\ g(f(x))) : I_{i+1}} \quad k \geq i-1$$

(Ssun-I'-1)

$$\frac{? \vdash g : I'_i}{? \vdash Letrec\ f\ Case(0,\ k,\ \ +(x,1),\ g(f(x))) : I'_{i+1}} \quad k \neq 0$$

(Ssun-I'-2)

$$\frac{? \vdash g : I''_i}{? \vdash Letrec\ f\ Case(0,\ 0,\ \ +(x,1),\ g(f(x))) : I'_{i+1}}$$

(Ssun-I"-1)

$$\frac{? \vdash g : I'_i}{? \vdash Letrec\ f\ Case(0,\ k,\ \ +(x,1),\ g(f(x))) : I''_{i+1}} \quad k > 1$$

(Ssun-I"-2)

$$\frac{? \vdash g : I''_i}{? \vdash Letrec\ f\ Case(0,\ 0,\ \ +(x,1),\ g(f(x))) : I''_{i+1}}$$

(Ssun-I"-3)

$$\frac{? \vdash g : I''_i}{? \vdash Letrec\ f\ Case(0,\ 1,\ \ +(x,1),\ g(f(x))) : I''_{i+1}}$$

(Sbin_1)

$$\frac{? \vdash g : s_1 \quad ? \vdash h : s_2}{? \vdash Letrec\ f\ Case((0,x_2),\ h(x_2),\ (+(x_1,1),x_2),\ g(f(x_1,x_2))) : B^1(s_1,s_2)}$$

(Sbin_2)

$$\frac{? \vdash h : s_1 \quad ? \vdash g : s_2}{? \vdash Letrec\ f\ Case((0,x_2),\ h(x_2),\ (+(x_1,1),x_2),\ f(x_1,g(x_2))) : B^2(s_1,s_2)}$$

(Sbin_3)

$$\frac{? \vdash g : s}{? \vdash Letrec\ f\ Case((0,x_2),\ k,\ (+(x_1,1),x_2),\ g(x_2,f(x_1,x_2))) : B^3(s)}$$

(Fonc)

$$?\,[f : s] \vdash f : s$$

Le coeur du système d'inférence est constitué des règles (Prim_1), (Prim_2), (Prim_2'), (Prim_2"), (Ssun-P), (Ssun-I), (Ssun-I'-1), (Ssun-I'-2), (Ssun-I"-1), (Ssun-I"-2), (Ssun-I"-3), (Sbin_1), (Sbin_2) et (Sbin_3). Les trois dernières correspondent aux définitions de B^1, B^2 et B^3 respectivement (cf. Définition 9) ; les autres règles reposent sur les définitions de P_i et I_i (cf. Définition 10). Les règles (P_1), (P_2) et (P_3) servent à propager l'inférence des schémas de fonctions au programme entier. Ce système de règles d'inférence conduit aussi à un algorithme puisque le choix entre des règles similaires est gouverné par k (qui est déterminé avant de chercher le type de g).

Chapitre 5

Concrétisation

La phase de concrétisation intervient après l'obtention d'un jeu de test abstrait. Celui-ci est dérivé de la plus petite borne supérieure des schémas associés au programme abstrait normalisé et à la propriété normalisée. La concrétisation est la dernière étape de notre méthode de test. Il s'agit en fait d'une étape optionnelle au sens où il est déjà possible, avec le jeu de test abstrait, de prouver qu'un programme vérifie une propriété donnée. Pour ce faire, il faut toutefois que le programme abstrait soit effectivement construit. On peut préférer tester directement le programme source. Il est alors nécessaire d'obtenir des valeurs du domaine standard qui correspondent au jeu de test abstrait. Si $J_a = \{v_a^1, \ldots, v_a^n\}$ est le jeu de test abstrait, on doit donc en dériver un jeu de test concret $J = \{v^1, \ldots, v^n\}$ tel que $\forall i \in \{1, \ldots, n\}$. $v^i \in Conc(v_a^i)$. Pour ce qui est de l'abstraction *longueur de liste*, il suffit donc de choisir, pour chaque entier v_a^i de J_a, une liste quelconque de taille v_a^i. Dans la pratique, on pourra préférer choisir des listes d'éléments distincts, qui possèdent un meilleur pouvoir discriminant et peuvent permettre de détecter des erreurs de programmation qui sortent du cadre de la propriété testée. C'est ce que fait le prototype décrit en Annexe D.

Reprenons l'exemple du programme *Reverse* et de la propriété *Id*. Nous savons que *Id* appartient au schéma S_1^1 (cf. Partie 2.2.2), et nous avons déterminé dans la Partie 4.2 que la version abstraite de *Reverse* appartient au schéma S_3^1. Comme expliqué dans le Chapitre 1, nous calculons le schéma commun à l'abstraction de *Reverse* et *Id* en prenant la plus petite borne supérieure des schémas respectifs de *Reverse* et *Id*. Le schéma commun est donc : $Lub(S_3^1, S_1^1) = S_3^1$. D'après la Propriété 18 de la Partie 2.2.2.3, $J_a = \{0, 1, 2, 3\}$ est un jeu de test abstrait complet pour vérifier que la longueur du résultat de *Reverse* est égale à la longueur de son argument.

Si on applique à J_a la procédure de concrétisation, on obtient par exemple le jeu de test concret : $J = \{nil, 0 :: nil, 1 :: 2 :: nil, 3 :: 4 :: 5 :: nil\}$. Ce jeu de test est ensuite utilisé sur le programme initial *Reverse* et on peut vérifier que la longueur du résultat rendu est bien égale à la longueur de l'argument, ce qui suffit à montrer la propriété recherchée en toute généralité.

Nous illustrons ce processus sur l'exemple du programme *Reverse* et de la pro-

priété Id.

Nous commençons par exécuter le jeu de test J sur le programme *Reverse*. On obtient les résultats suivants :

$$\begin{array}{lcl} Reverse(nil) & = & nil \\ Reverse(0::nil) & = & 0::nil \\ Reverse(1::2::nil) & = & 2::1::nil \\ Reverse(3::4::5::nil) & = & 5::4::3::nil \end{array}$$

Ensuite, nous exécutons le jeu de test abstrait J_a sur la propriété Id et nous obtenons :

$$\begin{array}{lcl} Id(0) & = & 0 \\ Id(1) & = & 1 \\ Id(2) & = & 2 \\ Id(3) & = & 3 \end{array}$$

La dernière étape consiste à vérifier que l'abstraction des résultats de l'exécution de *Reverse* sur J produit les mêmes valeurs que l'exécution de J_a sur Id. C'est effectivement le cas ici puisque :

$$\begin{array}{lcl} abs(nil) & = & 0 \\ abs(0::nil) & = & 1 \\ abs(2::1::nil) & = & 2 \\ abs(5::4::3::nil) & = & 3 \end{array}$$

À ce stade, notre méthode de test permet d'assurer que le programme *Reverse* rend effectivement, quel que soit son argument, une liste dont la longueur est égale à celle de son argument (*Reverse* vérifie la propriété Id).

Chapitre 6

Exemples d'application

Dans le corps du document nous avons utilisé le programme *Reverse* et la propriété *Id* pour expliciter les différentes étapes constituant notre méthode de test. Dans ce chapitre, nous présentons d'autres exemples pour l'illustrer. Comme dans le reste du document, nous considérons des fonctions sur les listes et nous cherchons à vérifier des propriétés sur les longueurs de leurs arguments et résultats. Nous utilisons donc la même abstraction que celle présentée dans le Chapitre 3. Nous commençons par un programme de remplacement (Partie 6.1), rendant une liste dont la longueur doit être le produit de la longueur de ses arguments. Puis nous continuons avec deux programmes de tri (Partie 6.2), qui sont supposés rendre en résultat une liste de la même longueur que leur argument.

6.1 Programme de remplacement

Considérons un programme *Rep* remplaçant chaque élément de son premier argument de type liste par son second argument. Ce programme peut s'écrire :

$$\begin{array}{lcl} Rep(nil, l) & = & nil \\ Rep(n::l_1, l_2) & = & Concat(l_2, Rep(l_1, l_2)) \end{array}$$

$$\begin{array}{lcl} Concat(nil, l) & = & l \\ Concat(n::l_1, l_2) & = & cons(n, Concat(l_1, l_2)) \end{array}$$

L'interprétation abstraite de *Rep* rend le programme :

$$\begin{array}{lcl} Lrep(0, n) & = & 0 \\ Lrep(n_1 + 1, n_2) & = & Lconcat(n_2, Lrep(n_1, n_2)) \end{array}$$

$$\begin{array}{lcl} Lconcat(0, m) & = & m \\ Lconcat(n + 1, m) & = & 1 + Lconcat(n, m) \end{array}$$

Après normalisation, on obtient le programme abstrait suivant :

$$\begin{array}{lcl} Lrep(0,n) & = & 0 \\ Lrep(n_1+1,n_2) & = & Lconcat(n_2, Lrep(n_1,n_2)) \\ \\ Lconcat(0,m) & = & m \\ Lconcat(n+1,m) & = & Lconcat(n,m)+1 \end{array}$$

L'inférence de schéma transforme d'abord ce programme en :

$$\begin{array}{lcl} Lrep(0,n) & = & 0 \\ Lrep(n_1+1,n_2) & = & Lconcat(n_2, Lrep(n_1,n_2)) \\ \\ Lconcat(0,m) & = & id(m) \\ Lconcat(n+1,m) & = & plus1(Lconcat(n,m)) \end{array}$$

et conclut que $Lrep$ est de schéma $B^3(B^1(I_1,I_1))$. En effet, le schéma $B^1(I_1,I_1)$ est associé à $Lconcat$ car id et $plus1$ sont de schéma I_1.

La propriété qui nous intéresse est que la longueur du résultat de Rep doit être égale au produit des longueurs de ses arguments, c'est-à-dire :

$$\begin{array}{lcl} Mult(0,m) & = & 0 \\ Mult(n+1,m) & = & Plus(m, Mult(n,m)) \\ \\ Plus(0,m) & = & m \\ Plus(n+1,m) & = & Plus(n,m)+1 \end{array}$$

L'inférence de schéma transforme ce programme en :

$$\begin{array}{lcl} Mult(0,m) & = & 0 \\ Mult(n+1,m) & = & Plus(m, Mult(n,m)) \\ \\ Plus(0,m) & = & id(m) \\ Plus(n+1,m) & = & plus1(Plus(n,m)) \end{array}$$

et détermine que $Mult$ est également de schéma $B^3(B^1(I_1,I_1))$ car $Plus$ est de schéma $B^1(I_1,I_1)$.

Donc, $J_a = \{(0,0),\ (1,1)\}$ est un jeu de test (abstrait) complet pour $Lrep$ et $Mult$. Après concrétisation, un jeu de test possible est :

$$J = \{(nil, nil),\ (1::nil, 2::nil)\}.$$

Il est alors suffisant de tester Rep avec J pour décider si la longueur de son résultat est bien le produit de la longueur de ses arguments.

6.2 Programmes de tri

Pour finir, nous considérons ici deux programmes de tri : le tri par sélection (Partie 6.2.1) et le tri par insertion (Partie 6.2.2). Dans ces deux cas, la propriété qui nous intéresse est que la longueur du résultat soit égale à la longueur de l'argument. La fonction abstraite est donc l'identité.

6.2.1 Tri par sélection

Un programme de tri par sélection peut être défini de la manière suivante :

$$\begin{array}{lcl} Selsort(nil) & = & nil \\ Selsort(n::l) & = & cons(Mine(n,l), Selsort(Restmin(n,l))) \\ \\ Mine(n, nil) & = & n \\ Mine(n, m::l) & = & Mine(Min(n,m), l) \\ \\ Restmin(n, nil) & = & nil \\ Restmin(n, m::l) & = & cons(Max(n,m), Restmin(n,l)) \\ \\ Min(n,m) & = & if \;\; n \leq m \;\; then \;\; n \;\; else \;\; m \\ \\ Max(n,m) & = & if \;\; n \geq m \;\; then \;\; n \;\; else \;\; m \end{array}$$

Après abstraction, on obtient le programme abstrait :

$$\begin{array}{lcl} Lselsort(0) & = & 0 \\ Lselsort(n+1) & = & 1 + Lselsort(Lrestmin(n)) \\ \\ Lrestmin(0) & = & 0 \\ Lrestmin(n+1) & = & 1 + Lrestmin(n) \end{array}$$

Remarquons que *Lrestmin* est d'arité 1 puisque *Restmin* n'a qu'un argument de type liste (l'autre étant de type entier).
Après normalisation, on obtient le programme abstrait normalisé :

$$\begin{array}{lcl} Lselsort(0) & = & 0 \\ Lselsort(n+1) & = & Lselsort(Lrestmin(n)) + 1 \\ \\ Lrestmin(0) & = & 0 \\ Lrestmin(n+1) & = & Lrestmin(n) + 1 \end{array}$$

L'inférence de schéma transforme ce programme en :

$$\begin{array}{lcl} Lselsort(0) & = & 0 \\ Lselsort(n+1) & = & plus1(Lselsort(Lrestmin(n))) \\ \\ Lrestmin(0) & = & 0 \\ Lrestmin(n+1) & = & plus1(Lrestmin(n)) \end{array}$$

et rend le schéma S_2^2 pour $Lselsort$ (le schéma S_2^1 étant associé à $Lrestmin$).

Donc, $J_a = \{0, 1, 2, 3, 4\}$ est un jeu de test complet pour décider si $Lselsort = Id$, et J_a peut se concrétiser en $J = \{nil, 0::nil, 1::2::nil, 3::4::5::nil, 6::7::8::9::nil\}$. Il est alors suffisant de tester le programme $Selsort$ sur J pour décider s'il rend bien une liste de même longueur que celle passée en argument.

Notons que nous garantissons uniquement la détection des fautes ayant un impact sur la longueur du résultat en utilisant ce jeu de test (puisque c'est précisément l'objectif de ce test). Imaginons par exemple que nous ayons oublié d'introduire la valeur $Max(n, m)$ dans le résultat de $Restmin$:

$$\begin{array}{lcl} Restmin(n, nil) & = & nil \\ Restmin(n, m::l) & = & Restmin(n, l) \end{array}$$

Cette erreur serait détectée par l'application de $Selsort$ à une liste de longueur 2. Mais si au lieu de cela nous avions remplacé Max par Min, l'erreur n'aurait pas nécessairement été détectée par un jeu de test concret J engendré de façon aléatoire à partir de J_a, à moins qu'il ne satisfasse la condition supplémentaire que les listes soient composées d'éléments distincts, ce qui est le cas pour la procédure de concrétisation utilisée dans notre système.

6.2.2 Tri par insertion

Le tri par insertion peut être défini de la manière suivante[1] :

$$\begin{array}{lcl} Insort(nil) & = & nil \\ Insort(n::l) & = & Insert(Insort(l), n) \\ \\ Insert(nil, m) & = & cons(m, nil) \\ Insert(n::l, m) & = & cons(Min(n, m), Insert(l, Max(n, m))) \end{array}$$

L'abstraction de ce programme rend le programme abstrait :

$$\begin{array}{lcl} Linsort(0) & = & 0 \\ Linsort(n+1) & = & Linsert(Linsort(n)) \\ \\ Linsert(0) & = & 1+0 \\ Linsert(n+1) & = & 1+Linsert(n) \end{array}$$

Le schéma inféré pour ce programme est S_3^1, donc $J_a = \{0, 1, 2, 3\}$ est un jeu de test complet pour décider si $Linsort = Id$. Le jeu de test J_a peut ensuite se concrétiser en $J = \{nil, 0::nil, 1::2::nil, 3::4::5::nil\}$. Ainsi, il est suffisant de tester le programme

1. En fait, une version plus efficace de $Insert$ pourrait être définie dans un langage fonctionnel, mais cela n'entre pas dans le cadre de nos schémas car ce n'est pas uniforme. Nous revenons sur ce problème en conclusion.

$Insort$ sur J pour décider s'il rend bien une liste de même longueur que celle passée en argument.

Pour illustrer la pertinence des jeux de test dérivés par notre méthode, considérons une définition erronée de $Insert$:

$$\begin{array}{lcl} Insort'(nil) & = & nil \\ Insort'(n::l) & = & Insert'(Insort'(l), n) \end{array}$$

$$\begin{array}{lcl} Insert'(nil, m) & = & cons(m, nil) \\ Insert'(n::l, m) & = & cons(Min(n, m), cons(Max(n, m), nil)) \end{array}$$

L'abstraction de $Insort'$ est :

$$\begin{array}{lcl} Linsort'(0) & = & 0 \\ Linsort'(n+1) & = & Linsert'(Linsort'(n)) \end{array}$$

$$\begin{array}{lcl} Linsert'(0) & = & 1 \\ Linsert'(n+1) & = & 2 \end{array}$$

L'inférence de schéma transforme ce programme en :

$$\begin{array}{lcl} Linsort'(0) & = & 0 \\ Linsort'(n+1) & = & Linsert'(Linsort'(n)) \end{array}$$

$$\begin{array}{lcl} Linsert'(0) & = & 1 \\ Linsert'(n+1) & = & \overline{2}(Linsert'(n)) \end{array}$$

$Linsert'$ appartient au schéma S_2^1 ; en conséquence $Linsort'$ et $Linsort$ appartiennent au même schéma S_3^1 et ont donc le même jeu de test complet.

Il se trouve que la définition erronée $Insort'$ rend des résultats corrects pour les listes de longueur inférieure ou égale à 2. Il est donc vraiment nécessaire d'inclure une liste de longueur 3 dans le jeu de test pour détecter cette erreur.

Conclusion

Bilan

La principale contribution de cette thèse est une nouvelle approche pour la vérification de programmes, intégrant les techniques de test et d'analyse statique. Nous proposons une méthode formelle de génération de jeux de test finis complets permettant de prouver qu'un programme vérifie une propriété donnée. Cette méthode utilise le texte du programme et de la propriété, qui doivent appartenir à certaines classes de programmes (ou de propriétés). Ces classes sont représentées par des hiérarchies de schémas, qui peuvent être vues comme des biais de test qui modélisent des hypothèses de test. Notre méthode est constituée de quatre étapes principales :

1. Abstraction du programme.
2. Inférence de deux schémas (l'un associé au programme, l'autre à la propriété).
3. Déduction d'un jeu de test abstrait pour le programme et la propriété.
4. Dérivation d'un jeu de test concret correspondant.

La dernière phase est facultative pour établir qu'un programme vérifie une propriété. Néanmoins, elle est utile d'un point de vue documentaire (consultation du jeu de test en vue d'ajouter de nouvelles données de test, ou d'évaluer sa couverture) et peut permettre de recueillir des informations supplémentaires sur le comportement du programme.

Pour une propriété donnée (c'est-à-dire une abstraction donnée), notre approche est complètement automatique. Elle ne nécessite donc aucune compétence particulière de l'utilisateur. Nous avons implanté cette méthode dans un prototype, pour le cas de la propriété *longueur*. Ce prototype accepte en entrée des programmes et des propriétés écrits dans un langage fonctionnel restreint.

La différence entre notre méthode et les techniques de vérification traditionnelles ou les méthodes formelles de développement (telles que Z [SPI92], VDM [JON90], B [ABR96], LARCH [GH93] ou Coq [PMW93]) est que nous privilégions la mécanisation par rapport à la généralité. Afin d'obtenir une méthode automatique, nous avons dû restreindre la classe de programmes et de propriétés considérés. Notons cependant que la restriction sur les programmes est plus faible que celle sur les propriétés (car ce ne sont que les versions abstraites des programmes qui doivent appartenir à un des schémas). À cause de cette restriction sur les propriétés, notre méthode peut être vue comme plus proche d'un vérificateur de type sophistiqué que d'une technique de vérification

classique. Par exemple, un vérificateur de type peut montrer qu'un programme rend un résultat de type *list*, alors que notre méthode fournit de l'information sur la longueur de cette liste.

Perspectives

Des expérimentations sont nécessaires pour mesurer l'impact des restrictions imposées par notre méthode et estimer les extensions nécessaires pour permettre une véritable utilisation pratique. Nous citons à présent un certain nombre de pistes de recherche qui ont déjà été identifiées.

Dans la thèse, nous n'avons étudié qu'une seule abstraction : celle de la fonction longueur. Un prolongement naturel consiste à intégrer d'autres propriétés s'exprimant dans le domaine des entiers (comme la taille ou la profondeur) et d'autres types de données que les listes et les entiers (comme les arbres ou les types inductifs généraux). Ces nouvelles abstractions s'expriment aussi naturellement dans le cadre de l'interprétation abstraite.

Une contrainte importante des schémas utilisés dans ce travail (et cruciale pour les preuves de nos résultats) est leur uniformité par rapport au type de données considéré (ici, les entiers naturels). Ce choix est inspiré par des travaux sur les types inductifs et les schémas de programmes inductifs qui leur sont associés [PDM89]. L'uniformité signifie que les choix dans un programme (ou une propriété) sont uniquement basés sur la structure des arguments. Par exemple, le programme $Insert$ présenté dans la Partie 6.2.2 est uniforme, mais il est possible d'en définir une autre version par :

$$\begin{array}{lcl} Insert(nil, m) & = & cons(m, nil) \\ Insert(n::l, m) & = & if \;\; m < n \;\; then \;\; cons(m, cons(n, l)) \;\; else \;\; cons(n, Insert(l, m)) \end{array}$$

qui n'est pas uniforme et n'entre donc pas dans le cadre de nos schémas. Une possibilité pour contourner cette limitation serait de dériver deux versions abstraites approximatives d'un programme, l'une représentant une borne inférieure et l'autre une borne supérieure de la propriété d'intérêt. D'un point de vue technique, il faudrait définir autant de fonctions d'abstraction que d'approximations désirées.

Par ailleurs, on pourrait étendre l'ensemble des programmes et propriétés traités en ajoutant de nouveaux éléments de langage. Une hiérarchie de schémas plus riche pourrait être utilisée directement comme langage de programmation (restreint). Cela pourrait être vu comme une discipline de programmation, fournissant une aide et un guide à la construction de programmes facilement testables. Une extension naturelle dans cette voie (et qui permettrait d'aller au-delà du domaine des entiers considéré jusqu'à présent) consiste à considérer des programmes et des propriétés polymorphes, le polymorphisme correspondant à une forme de régularité. Il faut d'ailleurs noter à ce sujet que l'idée d'exploiter des restrictions sur le langage de propriétés a aussi été explorée pour démontrer par le test des propriétés de sécurité de programmes Java [JLT98]. Un groupe de recherche américain [DAC98, DAC] a également proposé récemment une bibliothèque de modèles de propriétés en logique temporelle qui a permis de représenter, d'après leurs expériences, 92% des cas de spécifications rencontrés, dans le but de

faire de la vérification par *model checking*. Ces travaux montrent une convergence certaine autour de l'idée d'utiliser des formalismes restreints pour favoriser les vérifications automatiques de programmes.

De manière générale, nous pensons qu'il est possible de proposer des techniques intermédiaires entre celles qui sont très générales mais peu automatisées (comme les démonstrateurs de théorèmes qui nécessitent des utilisateurs qualifiés), et celles complètement automatiques mais de portée limitée (comme l'inférence de type traditionnelle). Cette thèse est une contribution dans ce sens, même s'il est clair qu'il reste du chemin à parcourir avant de déboucher sur des outils véritablement utilisables en pratique.

Annexes

Annexe A

Inférence de type

La syntaxe de notre langage de programmation L_{Prog} impose que chaque variable soit associée à un type (*List* ou bien type de base *Int*, *Real*...). L'objectif de l'inférence de type décrite ici est de fournir les informations nécessaires à l'abstraction décrite dans le Chapitre 3 de la Partie II. Pour cette abstraction, il suffit de pouvoir distinguer une variable de type *List* d'une variable d'un type de base. On ne considère donc que deux «types» ici :

- *base* qui regroupe *Int*, *Real*...
- *List* qui décrit les listes.

Les axiomes et règles d'inférence de type sont les suivants :

$$\frac{? \vdash Let\ op\ (x{:}t_1,\ e) : t_1 \to t_2 \quad ?\,[op : t_1 \to t_2] \vdash P : t_3}{? \vdash Let\ op\ (x{:}t_1,\ e); P : t_3}$$

$$\frac{? \vdash Letrec\ op\ Case(0,\ e_1,\ \ +(x{:}t_b, 1),\ e_2) : t_b \to t_1 \quad ?\,[op : t_b \to t_1] \vdash P : t_2}{? \vdash Letrec\ op\ Case(0,\ e_1,\ \ +(x{:}t_b, 1),\ e_2); P : t_2}$$

$$\frac{? \vdash Letrec\ op\ Case(nil,\ e_1,\ x{:}t_b{::}l{:}List,\ e_2) : List \to t_1 \quad ?\,[op : List \to t_1] \vdash P : t_2}{? \vdash Letrec\ op\ Case(nil,\ e_1,\ x{:}t_b{::}l{:}List,\ e_2); P : t_2}$$

$$\frac{\begin{array}{l}? \vdash Letrec\ op\ Case((0, x_1{:}t_1),\ e_1,\ (+(x_2{:}t_b, 1), x_3{:}t_1),\ e_2) : t_b \times t_1 \to t_2 \\ ?\,[op : t_b \times t_1 \to t_2] \vdash P : t_3\end{array}}{? \vdash Letrec\ op\ Case((0, x_1{:}t_1),\ e_1,\ (+(x_2{:}t_b, 1), x_3{:}t_1),\ e_2); P : t_3}$$

$$\frac{\begin{array}{l}? \vdash Letrec\ op\ Case((nil, x_1{:}t_1),\ e_1,\ (x_2{:}t_b{::}l{:}List, x_3{:}t_1),\ e_2) : List \times t_1 \to t_2 \\ ?\,[op : List \times t_1 \to t_2] \vdash P : t_3\end{array}}{? \vdash Letrec\ op\ Case((nil, x_1{:}t_1),\ e_1,\ (x_2{:}t_b{::}l{:}List, x_3{:}t_1),\ e_2); P : t_3}$$

$$\frac{?\,[x : t_1] \vdash e : t_2}{? \vdash Let\ op\ (x{:}t_1,\ e) : t_1 \to t_2}$$

$$\frac{? \vdash e_1 : t \quad ?[x : t_b,\ op : t_b \to t] \vdash e_2 : t}{? \vdash Letrec\ op\ Case(0,\ e_1,\ \ +(x{:}t_b, 1),\ e_2) : t_b \to t}$$

$$\frac{? \vdash e_1 : t \quad ?[x : t_b,\ l : List,\ op : List \to t] \vdash e_2 : t}{? \vdash Letrec\ op\ Case(nil,\ e_1,\ x{:}t_b{::}l{:}List,\ e_2) : List \to t}$$

$$\frac{?[x_1 : t_1] \vdash e_1 : t_2 \quad ?[x_2 : t_b,\ x_3 : t_1,\ op : t_b \times t_1 \to t_2] \vdash e_2 : t_2}{? \vdash Letrec\ op\ Case((0, x_1{:}t_1),\ e_1,\ (+(x_2{:}t_b, 1), x_3{:}t_1),\ e_2) : t_b \times t_1 \to t_2}$$

$$\frac{?[x_1 : t_1] \vdash e_1 : t_2 \quad ?[x_2 : t_b,\ l : List,\ x_3 : t_1,\ op : List \times t_1 \to t_2] \vdash e_2 : t_2}{? \vdash Letrec\ op\ Case((nil, x_1{:}t_1),\ e_1,\ (x_2{:}t_b{::}l{:}List, x_3{:}t_1),\ e_2) : List \times t_1 \to t_2}$$

$$\frac{? \vdash op : t_1 \to t_2 \quad ? \vdash e : t_1}{? \vdash op(e) : t_2}$$

$$\frac{? \vdash op : t_1 \times t_2 \to t_3 \quad ? \vdash e_1 : t_1 \quad ? \vdash e_2 : t_2}{? \vdash op(e_1, e_2) : t_3}$$

$$\frac{? \vdash e : Bool \quad ? \vdash e_1 : t \quad ? \vdash e_2 : t}{? \vdash if\ e\ then\ e_1\ else\ e_2 : t}$$

$? \vdash head : List \to t_b$ $\qquad$ $? \vdash tail : List \to List$

$? \vdash cons : t_b \times List \to List$

$? \vdash + : t_b \times t_b \to t_b$ $\qquad$ $? \vdash - : t_b \times t_b \to t_b$

$? \vdash * : t_b \times t_b \to t_b$ $\qquad$ $? \vdash / : t_b \times t_b \to t_b$

$? \vdash = : t_b \times t_b \to Bool$ $\qquad$ $? \vdash \neq : t_b \times t_b \to Bool$

$? \vdash > : t_b \times t_b \to Bool$ $\qquad$ $? \vdash < : t_b \times t_b \to Bool$

$? \vdash \geq : t_b \times t_b \to Bool$ $\qquad$ $? \vdash \leq : t_b \times t_b \to Bool$

$? \vdash nil : List$ $\qquad$ $? \vdash c : t_b$

$?[x : t] \vdash x : t$ $\qquad$ $?[op : t] \vdash op : t$

On peut remarquer que ce système conduit directement à un algorithme d'inférence car on peut lui associer un ordre strict dans le calcul des types, et il n'y a aucune ambiguïté dans le choix d'une règle à appliquer. Pour obtenir l'algorithme, il suffit de

lire les règles de bas en haut (conclusions vers hypothèses) et de gauche à droite. Par exemple, la septième règle se lit : «Pour prouver que l'expression

$$\mathit{Letrec}\ \mathit{op}\ \mathit{Case}(0,\ e_1,\ +(x{:}t_b, 1),\ e_2)$$

est de type $t_b \to t$ dans un contexte ?, il suffit de montrer que l'expression e_1 est de type t dans le contexte ?, et que l'expression e_2 est de type t dans le contexte ? augmenté de $[x : t_b,\ \mathit{op} : t_b \to t]$.». On remarque que t est connu grâce à la preuve de la première hypothèse avant d'être utilisé dans la preuve de la seconde.

Annexe B

Relation d'ordre associée à la normalisation

Rappelons tout d'abord le système de réécriture utilisé pour la normalisation dans la Partie 4.1 :

$$
\begin{array}{lcll}
+(e,0) & \rightarrow & e & \\
+(k_1,k_2) & \rightarrow & \overline{k_1+k_2} & \\
-(e,0) & \rightarrow & e & \\
-(k_1,k_2) & \rightarrow & \overline{k_1-k_2} & \\
+(k,e) & \rightarrow & +(e,k) & \text{avec } e \notin Const \\
+(+(e_1,e_2),e_3) & \rightarrow & +(e_1,+(e_2,e_3)) & \\
-(-(e_1,e_2),e_3) & \rightarrow & -(e_1,+(e_2,e_3)) & \\
-(+(e_1,e_2),e_3) & \rightarrow & +(e_1,-(e_2,e_3)) &
\end{array}
$$

On utilise la notation $\overline{a+b}$ pour dénoter le résultat de l'évaluation de $a+b$; e, e_1, e_2, e_3 pour dénoter une expression; k, k_1, k_2 pour dénoter une constante.
Les règles sont appliquées dans l'ordre où elles sont définies.

Nous montrons ci-après qu'il est possible d'associer un ordre bien fondé strictement décroissant à ce système de réécriture, ce qui nous permet de garantir sa terminaison (Propriétés 27 et 28).

Nous définissons maintenant cet ordre de réduction $<<$ pour la normalisation (Définition 14). Dans ce but, nous commençons par introduire deux notions de taille dans les Définitions 12 et 13.

Définition 12 *On définit* $taille_1$ *par :*

$$taille_1(e) = \sum_{k \epsilon e} poids(pos_e(k), e)$$

avec :

$$\begin{array}{lcl} pos_e(k) & = & \textit{position de } k \textit{ dans } e \\ poids(position, e) & = & 1 + \textit{nombre de symboles (constantes ou variables)} \\ & & \textit{à droite de position dans } e \end{array}$$

Définition 13 *On définit* $taille_2$ *par :*

$$\begin{array}{lcl} taille_2(k) & = & 1 \\ taille_2(x) & = & 1 \\ taille_2(+(e_1, e_2)) & = & 2 * taille_2(e_1) + taille_2(e_2) \\ taille_2(-(e_1, e_2)) & = & 2 * taille_2(e_1) + taille_2(e_2) \end{array}$$

Définition 14 *On définit l'ordre* $<<$ *par :*

$$\begin{array}{lcl} e_1 << e_2 & \Longleftrightarrow & taille_1(e_1) < taille_1(e_2) \\ & & \vee \\ & & taille_1(e_1) = taille_1(e_2) \wedge taille_2(e_1) < taille_2(e_2) \end{array}$$

Nous pouvons à présent établir la propriété suivante :

Propriété 27
Toutes les règles de réécriture pour la normalisation font décroître strictement les expressions selon l'ordre $<<$.

Preuve

Soient $nbs(e) \geq 0$ le nombre de symboles dans e, et $nbc(e) \geq 0$ le nombre de constantes dans e.

Les cinq premières règles font décroître strictement la valeur de taille_1.

• Première règle : $+(e, 0) \rightarrow e$.

$$\begin{array}{lcl} \text{taille}_1(+(e,0)) & = & 1 + \sum_{k \epsilon e} \text{poids}(\text{pos}_{+(e,0)}(k), +(e,0)) \\ & = & 1 + \sum_{k \epsilon e} \text{poids}(\text{pos}_e(k), +(e,0)) \\ & = & 1 + \sum_{k \epsilon e} (\text{poids}(\text{pos}_e(k), e) + 1) \\ & = & 1 + nbc(e) + \sum_{k \epsilon e} \text{poids}(\text{pos}_e(k), e) \\ & = & 1 + nbc(e) + \text{taille}_1(e) \end{array}$$

Donc, $\text{taille}_1(e) < \text{taille}_1(+(e, 0))$.

• Seconde règle : $+(k_1, k_2) \rightarrow \overline{k_1 + k_2}$.

$$\begin{array}{lcl} \text{taille}_1(+(k_1, k_2)) & = & 2 + 1 \quad = \quad 3 \\ \text{taille}_1(\overline{k_1 + k_2}) & = & 1 \end{array}$$

Donc, $\text{taille}_1(\overline{k_1 + k_2}) < \text{taille}_1(+(k_1, k_2))$.

- Troisième règle : $-(e,0) \to e$.

$$\begin{array}{rcl}
\text{taille}_1(-(e,0)) & = & 1 + \sum_{k \epsilon e} \text{poids}(\text{pos}_{-(e,0)}(k), -(e,0)) \\
& = & 1 + \sum_{k \epsilon e} \text{poids}(\text{pos}_e(k), -(e,0)) \\
& = & 1 + \sum_{k \epsilon e} (\text{poids}(\text{pos}_e(k), e) + 1) \\
& = & 1 + nbc(e) + \sum_{k \epsilon e} \text{poids}(\text{pos}_e(k), e) \\
& = & 1 + nbc(e) + \text{taille}_1(e)
\end{array}$$

Donc, $\text{taille}_1(e) < \text{taille}_1(-(e,0))$.

- Quatrième règle : $-(k_1,k_2) \to \overline{k_1 - k_2}$.

$$\begin{array}{rcl}
\text{taille}_1(-(k_1,k_2)) & = & 2+1 \quad = \quad 3 \\
\text{taille}_1(\overline{k_1 - k_2}) & = & 1
\end{array}$$

Donc, $\text{taille}_1(\overline{k_1 - k_2}) < \text{taille}_1(-(k_1,k_2))$.

- Cinquième règle : $+(k,e) \to +(e,k)$.

$$\begin{array}{rcl}
\text{taille}_1(+(k,e)) & = & 1 + nbs(e) + \sum_{c \epsilon e} \text{poids}(\text{pos}_{+(k,e)}(c), +(k,e)) \\
& = & 1 + nbs(e) + \sum_{c \epsilon e} \text{poids}(\text{pos}_e(c), e) \\
& = & 1 + nbs(e) + \text{taille}_1(e)
\end{array}$$

$$\begin{array}{rcl}
\text{taille}_1(+(e,k)) & = & 1 + \sum_{c \epsilon e} \text{poids}(\text{pos}_{+(e,k)}(c), +(e,k)) \\
& = & 1 + \sum_{c \epsilon e} \text{poids}(\text{pos}_e(c), +(e,k)) \\
& = & 1 + \sum_{c \epsilon e} (\text{poids}(\text{pos}_e(c), e) + 1) \\
& = & 1 + nbc(e) + \sum_{c \epsilon e} \text{poids}(\text{pos}_e(c), e) \\
& = & 1 + nbc(e) + \text{taille}_1(e)
\end{array}$$

Or, $nbc(e) < nbs(e)$ (car sinon e ne contiendrait que des constantes, et on aurait appliqué une des règles précédentes), donc $\text{taille}_1(+(e,k)) < \text{taille}_1(+(k,e))$.

Les trois dernières règles maintiennent la valeur de taille_1 constante, et font décroître strictement la valeur de taille_2.

- Sixième règle : $+(+(e_1,e_2),e_3) \to +(e_1,+(e_2,e_3))$.

$$\text{taille}_1(+(+(e_1,e_2),e_3)) = \text{taille}_1(+(e_1,+(e_2,e_3)))$$

$$\begin{array}{rcl}
\text{taille}_2(+(+(e_1,e_2),e_3)) & = & 2 * \text{taille}_2(+(e_1,e_2)) + \text{taille}_2(e_3) \\
& = & 4 * \text{taille}_2(e_1) + 2 * \text{taille}_2(e_2) + \text{taille}_2(e_3)
\end{array}$$

$$\begin{array}{rcl}
\text{taille}_2(+(e_1,+(e_2,e_3))) & = & 2 * \text{taille}_2(e_1) + \text{taille}_2(+(e_2,e_3)) \\
& = & 2 * \text{taille}_2(e_1) + 2 * \text{taille}_2(e_2) + \text{taille}_2(e_3)
\end{array}$$

Et $\text{taille}_2(+(e_1,+(e_2,e_3))) < \text{taille}_2(+(+(e_1,e_2),e_3))$ car on a $\text{taille}_2(e_1) > 0$ d'après la Définition 13.

• Septième règle : $-(-(e_1,e_2),e_3) \rightarrow -(e_1,+(e_2,e_3))$.

$$\begin{aligned}
\text{taille}_1(-(-(e_1,e_2),e_3)) &= \text{taille}_1(-(e_1,+(e_2,e_3)))\\
\\
\text{taille}_2(-(-(e_1,e_2),e_3)) &= 2 * \text{taille}_2(-(e_1,e_2)) + \text{taille}_2(e_3)\\
&= 4 * \text{taille}_2(e_1) + 2 * \text{taille}_2(e_2) + \text{taille}_2(e_3)\\
\\
\text{taille}_2(-(e_1,+(e_2,e_3))) &= 2 * \text{taille}_2(e_1) + \text{taille}_2(+(e_2,e_3))\\
&= 2 * \text{taille}_2(e_1) + 2 * \text{taille}_2(e_2) + \text{taille}_2(e_3)
\end{aligned}$$

Et $\text{taille}_2(-(e_1,+(e_2,e_3))) < \text{taille}_2(-(-(e_1,e_2),e_3))$ car on a $\text{taille}_2(e_1) > 0$ d'après la Définition 13.

• Huitième règle : $-(+(e_1,e_2),e_3) \rightarrow +(e_1,-(e_2,e_3))$.

$$\begin{aligned}
\text{taille}_1(-(+(e_1,e_2),e_3)) &= \text{taille}_1(+(e_1,-(e_2,e_3)))\\
\\
\text{taille}_2(-(+(e_1,e_2),e_3)) &= 2 * \text{taille}_2(+(e_1,e_2)) + \text{taille}_2(e_3)\\
&= 4 * \text{taille}_2(e_1) + 2 * \text{taille}_2(e_2) + \text{taille}_2(e_3)\\
\\
\text{taille}_2(+(e_1,-(e_2,e_3))) &= 2 * \text{taille}_2(e_1) + \text{taille}_2(-(e_2,e_3))\\
&= 2 * \text{taille}_2(e_1) + 2 * \text{taille}_2(e_2) + \text{taille}_2(e_3)
\end{aligned}$$

Et $\text{taille}_2(+(e_1,-(e_2,e_3))) < \text{taille}_2(-(+(e_1,e_2),e_3))$ car on a $\text{taille}_2(e_1) > 0$ d'après la Définition 13.
□

La Propriété 28 nous permet finalement de conclure :

Propriété 28
L'ordre $<<$ est bien fondé, les règles de normalisation forment donc un système convergent.

Annexe C

Preuves des Propriétés 23 et 24 (schémas binaires)

Propriété 24
$(\{0\} \times \{0, \ldots, i-1\}) \cup (\{1\} \times \{0, \ldots, j-1\})$ *est un jeu de test complet pour* $B^2(I_i, I_j)$.
$(\{0\} \times \{0, \ldots, i-1\}) \cup (\{1\} \times \{0, \ldots, j\})$ *est un jeu de test complet pour* $B^2(I_i, P_j)$.

Preuve
• **Schéma** $B^2(I_i, I_j)$

Soient f et f' appartenant à $B^2(I_i, I_j)$ et

$$\forall (x, y) \in (\{0\} \times \{0, \ldots, i-1\}) \cup (\{1\} \times \{0, \ldots, j-1\}) \ . \ f(x, y) = f'(x, y).$$

On a donc

$$\begin{array}{lcl} f(0, m) & = & h(m) \\ f(n+1, m) & = & f(n, g(m)) \end{array}$$

et

$$\begin{array}{lcl} f'(0, m) & = & h'(m) \\ f'(n+1, m) & = & f'(n, g'(m)) \end{array}$$

avec $h \in I_i$, $h' \in I_i$, $g \in I_j$ et $g' \in I_j$.
La propriété

$$\forall (x, y) \in \{0\} \times \{0, \ldots, i-1\} \ . \ f(x, y) = f'(x, y)$$

est équivalente à

$$\forall y \in \{0, \ldots, i-1\} \ . \ h(y) = h'(y)$$

ce qui implique $h = h'$ d'après les Propriétés 4 et 17 (puisque $h \in I_i$ et $h' \in I_i$).
La propriété

$$\forall (x,y) \in \{1\} \times \{0,\ldots,j-1\} \;.\; f(x,y) = f'(x,y)$$

est équivalente à

$$\forall y \in \{0,\ldots,j-1\} \;.\; h(g(y)) = h'(g'(y)) = h(g'(y))$$

ce qui implique

$$\forall y \in \{0,\ldots,j-1\} \;.\; g(y) = g'(y)$$

car h est injective (puisque croissante). De nouveau, la Propriété 4 nous permet de conclure $g = g'$. De $h = h'$ et $g = g'$, nous déduisons $f = f'$.

• **Schéma** $B^2(I_i, P_j)$

Soient f et f' appartenant à $B^2(I_i, P_j)$ et

$$\forall (x,y) \in (\{0\} \times \{0,\ldots,i-1\}) \cup (\{1\} \times \{0,\ldots,j\}) \;.\; f(x,y) = f'(x,y).$$

On a donc

$$\begin{array}{lcl} f(0,m) & = & h(m) \\ f(n+1,m) & = & f(n,g(m)) \end{array}$$

et

$$\begin{array}{lcl} f'(0,m) & = & h'(m) \\ f'(n+1,m) & = & f'(n,g'(m)) \end{array}$$

avec $h \in I_i$, $h' \in I_i$, $g \in P_j$ et $g' \in P_j$.

La propriété

$$\forall (x,y) \in \{0\} \times \{0,\ldots,i-1\} \;.\; f(x,y) = f'(x,y)$$

est équivalente à

$$\forall y \in \{0,\ldots,i-1\} \;.\; h(y) = h'(y)$$

ce qui implique $h = h'$ d'après la Propriété 4.

La propriété

$$\forall (x,y) \in \{1\} \times \{0,\ldots,j\} \;.\; f(x,y) = f'(x,y)$$

est équivalente à

$$\forall y \in \{0,\ldots,j\} \;.\; h(g(y)) = h'(g'(y)) = h(g'(y))$$

ce qui implique

$$\forall y \in \{0, \ldots, j\} \ . \ g(y) = g'(y)$$

car h est injective (puisque croissante). La Propriété 3 nous permet de conclure $g = g'$. De $h = h'$ et $g = g'$, nous pouvons déduire $f = f'$.

□

Propriété 25
$\{(0,0)\} \cup (\{1\} \times \{1, \ldots, i\}) \cup (\{1, \ldots, j\} \times \{1\})$ *est un jeu de test complet pour* $B^3(B^1(I_i, I_j))$.

Preuve
Soit f appartenant à $B^3(B^1(I_i, I_j))$. On a donc

$$\begin{array}{lcl} f(0,m) & = & k \\ f(n+1,m) & = & g(m, f(n,m)) \end{array}$$

et

$$\begin{array}{lcl} g(0,x) & = & h'(x) \\ g(m+1,x) & = & g'(g(m,x)) \end{array}$$

avec $g' \in I_i$, et $h' \in I_j$.
Pour plus de clarté, on définit :

$$f(n,m) = g_m^n(k)$$

avec

$$g_m(x) = g(m,x)$$

et on utilise :

$$g(m,x) = g'^m(h'(x)).$$

Pour déterminer la valeur k, il suffit de tester f avec une paire $(0,m)$, m étant quelconque, car $\forall m \in \mathbb{N} \ . \ f(0,m) = k$.

À présent, nous considérons deux cas :

1. $h'(k) \geq \theta_{g'}$,
2. $h'(k) < \theta_{g'}$.

• Premier cas : $h'(k) \geq \theta_{g'}$.

◇ Si $n = 1$ alors

$$f(1,m) = g(m,k) = g'^m(h'(k)).$$

Comme g' appartient à I_i, on peut la déterminer (cf. Propriété 4) en la testant avec les valeurs $m \in \{1, \ldots, i\}$. En effet, toutes les valeurs $h'(k), \ldots, g'^{i-1}(h'(k))$ sont distinctes puisque $h'(k) \geq \theta_{g'}$.

◇ Si $m = 1$ alors

$$f(n,1) = g(1, g(1, \ldots (g(1,k)) \ldots)) = (g' \circ h')^n(k).$$

Comme la fonction h' est croissante (puisque $h' \in I_j$) et $h'(k) \geq \theta_{g'}$, toutes les valeurs $k, g'(h'(k)), \ldots, (g' \circ h')^{j-1}(k))$ sont distinctes.
De plus, g' est croissante donc injective (car $g' \in I_i$), on peut donc déterminer h' en testant f sur les valeurs $n \in \{1, \ldots, j\}$.

• Second cas : $h'(k) < \theta_{g'}$.

◇ Si $n = 1$ alors

$$f(1,m) = g(m,k) = g'^m(h'(k)) = h'(k).$$

Avec m quelconque, on détermine $h'(k)$.

Nous considérons maintenant deux autres cas :
◇ Cas A : $k < \theta_{h'}$.
On a donc $g'(h'(k)) = h'(k) = k$, d'où :

$$\forall n \in \mathbb{N} \ . \ \forall m \in \mathbb{N} \ . \ f(n,m) = k.$$

◇ Cas B : $k \geq \theta_{h'}$.
Si $m = 1$ alors

$$f(n,1) = g(1, g(1, \ldots (g(1,k)) \ldots)) = (g' \circ h')^n(k).$$

Comme la fonction g' est croissante (puisque $g' \in I_i$) et $k \geq \theta_{h'}$, toutes les valeurs $k, g'(h'(k)), \ldots, (g' \circ h')^{j-1}(k))$ sont distinctes.
De plus, g' est injective (puisque croissante), donc on peut déterminer h' en testant f sur les valeurs $n \in \{1, \ldots, j\}$.

Nous avons donc montré que

$$\{(0,0)\} \cup (\{1\} \times \{1, \ldots, i\}) \cup (\{1, \ldots, j\} \times \{1\}) \ . \ f(x,y) = f'(x,y)$$

est un jeu de test suffisant pour déterminer f.
□

Annexe D

Implantation

Nous avons réalisé en Caml Light un prototype implantant les résultats théoriques décrits dans la thèse, pour le cas de l'abstraction *longueur*. Il permet en particulier de traiter tous les exemples cités dans la thèse. Ce prototype prend en entrée un programme et une propriété écrits dans des langages fonctionnels restreints (cf. ci-après), et calcule un jeu de test (concret) permettant de prouver que le programme vérifie la propriété. La Figure D.1 présente la structure du prototype.

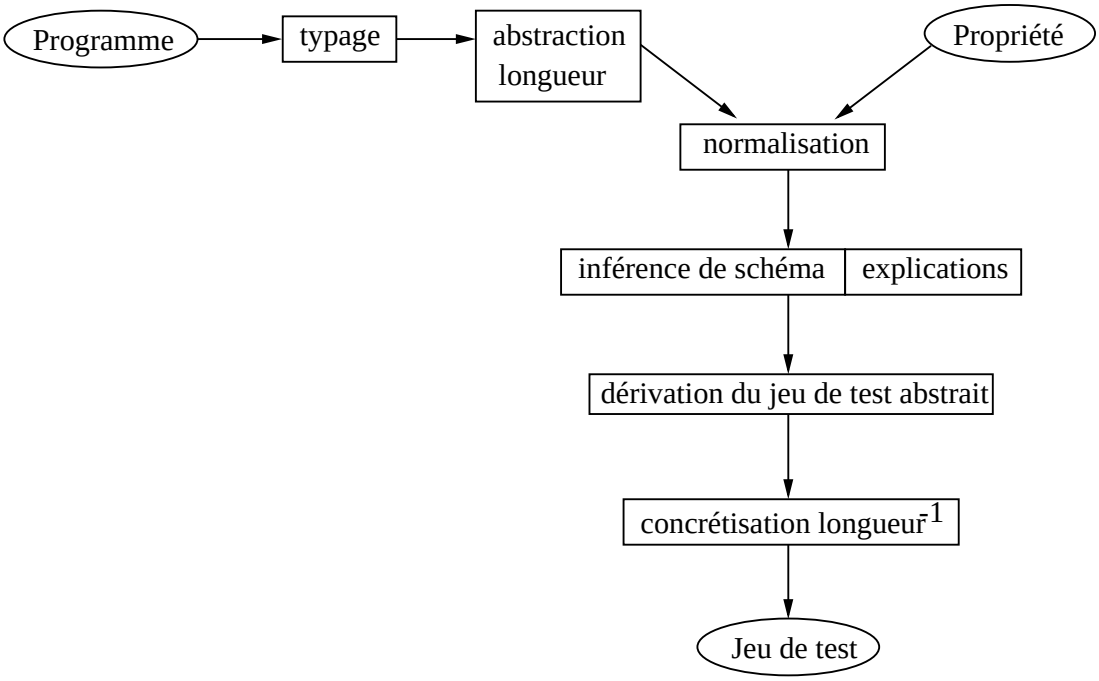


FIG. D.1 – *Structure du prototype*

Cette annexe contient quelques détails techniques concernant la réalisation du prototype.

Langage de programmation L_{Prog} - syntaxe concrète :

$$
\begin{array}{lcl}
Prog & = & Def\ ;\ Prog \mid \\
 & & Def. \\
Def & = & Let\ Fun\ (Var{:}Type,\ Exp) \mid \\
 & & Letrec\ Fun\ Case(0,\ Exp,\ Add_1,\ Exp) \mid \\
 & & Letrec\ Fun\ Case(nil,\ Exp,\ Var{:}Type{::}Var{:}List,\ Exp) \mid \\
 & & Letrec\ Fun\ Case((0,Var{:}Type),\ Exp,\ (Add_1,Var{:}Type),\ Exp) \mid \\
 & & Letrec\ Fun\ Case((nil,Var{:}Type),\ Exp, \\
 & & \qquad\qquad (Var{:}Type{::}Var{:}List,Var{:}Type),\ Exp) \\
Exp & = & OP_1(Exp) \mid \\
 & & OP_2(Exp,Exp) \mid \\
 & & if\ \ Exp\ \ then\ \ Exp\ \ else\ \ Exp \mid \\
 & & Var \mid \\
 & & Const \\
OP_1 & = & Fun \mid head \mid tail \\
OP_2 & = & Fun \mid cons \mid + \mid - \mid * \mid / \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid \ldots \\
Const & = & nil \mid 0 \mid 1 \mid \ldots \\
Add_1 & = & +(Var{:}Type,1) \mid +(1,Var{:}Type) \\
Type & = & Int \mid List
\end{array}
$$

Le non-terminal Fun dénote l'ensemble des identificateurs de fonction et le non-terminal Var l'ensemble des variables (du premier ordre).

Langage de programmation - syntaxe abstraite :

$$
\begin{array}{lcl}
Prog & = & Def\ ;\ Prog \mid \\
 & & Def \\
Def & = & abs(Fun,Var,Type,Exp) \mid \\
 & & recint_1(Fun,Exp,Var,Type,Exp) \mid \\
 & & reclist_1(Fun,Exp,Var,Type,Var,List,Exp) \mid \\
 & & recint_2(Fun,Var,Type,Exp,Var,Type,Var,Type,Exp) \mid \\
 & & reclist_2(Fun,Var,Type,Exp,Var,Type,Var,List,Var,Type,Exp) \\
Exp & = & app_1(OP_1,Exp) \mid \\
 & & app_2(OP_2,Exp,Exp) \mid \\
 & & cond(Exp,Exp,Exp) \mid \\
 & & Var \mid \\
 & & Const \\
OP_1 & = & Fun \mid head \mid tail \\
OP_2 & = & Fun \mid cons \mid plus \mid moins \mid mult \mid div \mid egal \mid diff \mid sup \mid inf \mid supeg \mid infeg \mid \ldots \\
Const & = & nil \mid 0 \mid 1 \mid \ldots \\
Type & = & Int \mid List
\end{array}
$$

Langage de propriétés L_{Prop} (ou abstrait) - syntaxe concrète :

$$
\begin{array}{lcl}
Prog & = & Def\ ;\ Prog\ \mid \\
 & & Def. \\
Def & = & Let\ Fun\ (Var,\ Exp)\ \mid \\
 & & Letrec\ Fun\ Case(0,\ Exp,\ Add_1,\ Exp)\ \mid \\
 & & Letrec\ Fun\ Case((0, Var),\ Exp,\ (Add_1, Var),\ Exp) \\
Exp & = & OP_1(Exp)\ \mid \\
 & & OP_2(Exp, Exp)\ \mid \\
 & & Var\ \mid \\
 & & Const \\
OP_1 & = & Fun \\
OP_2 & = & Fun \mid +\ \mid -\ \mid \ldots \\
Const & = & 0 \mid 1 \mid \ldots \\
Add_1 & = & +(Var, 1)\ \mid\ +(1, Var)
\end{array}
$$

Langage de propriété (ou abstrait) - syntaxe abstraite :

$$
\begin{array}{lcl}
Prog & = & Def\ ;\ Prog\ \mid \\
 & & Def \\
Def & = & abs(Fun, Var, Exp)\ \mid \\
 & & recint_1(Fun, Exp, Var, Exp)\ \mid \\
 & & recint_2(Fun, Var, Exp, Var, Var, Exp) \\
Exp & = & app_1(OP_1, Exp)\ \mid \\
 & & app_2(OP_2, Exp, Exp)\ \mid \\
 & & Var\ \mid \\
 & & Const \\
OP_1 & = & Fun \\
OP_2 & = & Fun \mid plus \mid moins \mid \ldots \\
Const & = & 0 \mid 1 \mid \ldots
\end{array}
$$

D.1 Module d'inférence de type

On utilise deux types :

– les entiers naturels Int,

– $List$ qui décrit les listes.

Les axiomes et règles d'inférence de type sont les suivants :

$$\frac{? \vdash abs(op, x, t_1, e) : t_1 \to t_2 \quad ?\,[op : t_1 \to t_2] \vdash P : t_3}{? \vdash abs(op, x, t_1, e); P : t_3}$$

$$\frac{? \vdash recint_1(op, e_1, x, Int, e_2) : Int \to t_1 \quad ?\,[op : Int \to t_1] \vdash P : t_2}{? \vdash recint_1(op, e_1, x, Int, e_2); P : t_2}$$

$$\frac{? \vdash reclist_1(op, e_1, x, Int, l, List, e_2) : List \to t_1 \quad ?[op : List \to t_1] \vdash P : t_2}{? \vdash reclist_1(op, e_1, x, Int, l, List, e_2); P : t_2}$$

$$\frac{? \vdash recint_2(op, x_1, t_1, e_1, x_2, Int, x_3, t_1, e_2) : Int \times t_1 \to t_2 \quad ?[op : Int \times t_1 \to t_2] \vdash P : t_3}{? \vdash recint_2(op, x_1, t_1, e_1, x_2, Int, x_3, t_1, e_2); P : t_3}$$

$$\frac{? \vdash reclist_2(op, x_1, t_1, e_1, x_2, Int, l, List, x_3, t_1, e_2) : List \times t_1 \to t_2 \quad ?[op : List \times t_1 \to t_2] \vdash P : t_3}{? \vdash reclist_2(op, x_1, t_1, e_1, x_2, Int, l, List, x_3, t_1, e_2); P : t_3}$$

$$\frac{?[x : t_1] \vdash e : t_2}{? \vdash abs(op, x, t_1, e) : t_1 \to t_2}$$

$$\frac{? \vdash e_1 : t \quad ?[x : Int,\ op : Int \to t] \vdash e_2 : t}{? \vdash recint_1(op, e_1, x, Int, e_2) : Int \to t}$$

$$\frac{? \vdash e_1 : t \quad ?[x : Int,\ l : List,\ op : List \to t] \vdash e_2 : t}{? \vdash reclist_1(op, e_1, x, Int, l, List, e_2) : List \to t}$$

$$\frac{?[x_1 : t_1] \vdash e_1 : t_2 \quad ?[x_2 : Int,\ x_3 : t_1,\ op : Int \times t_1 \to t_2] \vdash e_2 : t_2}{? \vdash recint_2(op, x_1, t_1, e_1, x_2, Int, x_3, t_1, e_2) : Int \times t_1 \to t_2}$$

$$\frac{?[x_1 : t_1] \vdash e_1 : t_2 \quad ?[x_2 : Int,\ l : List,\ x_3 : t_1,\ op : List \times t_1 \to t_2] \vdash e_2 : t_2}{? \vdash reclist_2(op, x_1, t_1, e_1, x_2, Int, l, List, x_3, t_1, e_2) : List \times t_1 \to t_2}$$

$$\frac{? \vdash op : t_1 \to t_2 \quad ? \vdash e : t_1}{? \vdash app_1(op, e) : t_2}$$

$$\frac{? \vdash op : t_1 \times t_2 \to t_3 \quad ? \vdash e_1 : t_1 \quad ? \vdash e_2 : t_2}{? \vdash app_2(op, e_1, e_2) : t_3}$$

$$\frac{? \vdash e : Bool \quad ? \vdash e_1 : t \quad ? \vdash e_2 : t}{? \vdash cond(e, e_1, e_2) : t}$$

$? \vdash head : List \to Int$ $\quad$ $? \vdash tail : List \to List$

$? \vdash cons : Int \times List \to List$

$? \vdash plus : Int \times Int \to Int$ $\quad$ $? \vdash moins : Int \times Int \to Int$

$? \vdash mult : Int \times Int \to Int$ $\quad$ $? \vdash div : Int \times Int \to Int$

$? \vdash egal : Int \times Int \to Bool$ $\quad$ $? \vdash diff : Int \times Int \to Bool$

$? \vdash sup : Int \times Int \to Bool$ $\quad$ $? \vdash inf : Int \times Int \to Bool$

$? \vdash supeg : Int \times Int \to Bool$ $\quad$ $? \vdash infeg : Int \times Int \to Bool$

$? \vdash nil : List$ $\quad$ $? \vdash c : Int$

$?[x : t] \vdash x : t$ $\quad$ $?[op : t] \vdash op : t$

D.2 Module d'abstraction

L'abstraction est mise en oeuvre par la transformation T_a suivante:

$$
\begin{array}{lcl}
T_a(F;P) & = & T_a(F);T_a(P) \\
T_a(abs(f,x,List,e)) & = & abs(f,x,T_a(e)) \\
T_a(reclist_1(f,e_1,x,Int,l,List,e_2)) & = & recint_1(f,T_a(e_1),l,T_a(e_2)) \\
T_a(reclist_2(f,x,Int,e_1,x_1,Int,l,List,x_2,Int,e_2)) & = & recint_1(f,T_a(e_1),l,T_a(e_2)) \\
T_a(reclist_2(f,l,List,e_1,x_1,Int,l_1,List,l_2,List,e_2)) & = & recint_2(f,l,T_a(e_1),l_1,l_2,T_a(e_2)) \\
T_a(app_1(f_{List\to List},e)) & = & app_1(f,T_a(e)) \\
T_a(app_1(tail,l)) & = & app_2(moins,T_a(l),1) \\
T_a(app_2(f_{List\times List\to List},e_1,e_2)) & = & app_2(f,T_a(e_1),T_a(e_2)) \\
T_a(app_2(f_{List\times Int\to List},e_1,e_2)) & = & app_1(f,T_a(e_1)) \\
T_a(app_2(f_{Int\times List\to List},e_1,e_2)) & = & app_1(f,T_a(e_2)) \\
T_a(app_2(cons,x,l)) & = & app_2(plus,T_a(l),1) \\
T_a(x_{List}) & = & x \\
T_a(nil) & = & 0
\end{array}
$$

D.3 Module de normalisation

La normalisation est mise en oeuvre par la transformation T_n suivante :

$$
\begin{array}{lcl}
T_n(F;P) & = & T_n(F);T_n(P) \\
T_n(abs(f,x,e)) & = & abs(f,x,T_n(e)) \\
T_n(recint_1(f,e_1,x,e_2)) & = & recint_1(f,T_n(e_1),x,T_n(e_2)) \\
T_n(recint_2(f,x_1,e_1,x_2,x_3,e_2)) & = & recint_2(f,x_1,T_n(e_1),x_2,x_3,T_n(e_2)) \\
T_n(app_2(op,e_1,e_2)) & = & Norm(app_2(op,T_n(e_1),T_n(e_2))) \\
T_n(app_1(op,e)) & = & app_1(op,T_n(e)) \\
T_n(e) & = & e \\
Norm(app_2(plus,e,0)) & = & e \\
Norm(app_2(plus,k_1,k_2)) & = & \overline{k_1+k_2} \\
Norm(app_2(moins,e,0)) & = & e \\
Norm(app_2(moins,k_1,k_2)) & = & \overline{k_1-k_2} \\
Norm(app_2(plus,k,x)) & = & app_2(plus,x,k) \\
Norm(app_2(plus,k,app_1(op,e))) & = & app_2(plus,app_1(op,e),k) \\
Norm(app_2(plus,k,app_2(op,e_1,e_2))) & = & app_2(plus,app_2(op,e_1,e_2),k) \\
Norm(app_2(plus,app_2(plus,e_1,e_2),e_3)) & = & app_2(plus,e_1,app_2(plus,e_2,e_3)) \\
Norm(app_2(moins,app_2(moins,e_1,e_2),e_3)) & = & app_2(moins,e_1,app_2(plus,e_2,e_3)) \\
Norm(app_2(moins,app_2(plus,e_1,e_2),e_3)) & = & app_2(plus,e_1,app_2(moins,e_2,e_3)) \\
Norm(app_2(op,e_1,e_2)) & = & app_2(op,T_n(e_1),Norm(e_2)) \\
Norm(e) & = & e
\end{array}
$$

D.4 Module d'inférence de schémas unaires

Les axiomes et règles d'inférence de schémas unaires sont les suivants :

(P_1)
$$\frac{? \vdash abs(f,x,e) : s_1 \quad ?[f:s_1] \vdash P : s_2}{? \vdash abs(f,x,e);P : s_2}$$

(P_2)
$$\frac{? \vdash recint_1(f,e_1,x,e_2) : s_1 \quad ?[f:s_1] \vdash P : s_2}{? \vdash recint_1(f,e_1,x,e_2);P : s_2}$$

($Prim_1$)
$$? \vdash abs(f,x,x) : S_1^1$$

($Prim_2$)
$$? \vdash abs(f,x,k) : S_1^1$$

($Prim_3$)
$$? \vdash abs(f,x,app_2(plus,x,k)) : S_1^1$$

(Ssun)
$$\frac{? \vdash g : S_i^1}{? \vdash recint_1(f,k,x,app_1(g,app_1(f,x))) : S_{i+1}^1}$$

(Scun)

$$\frac{? \vdash g : S^1_{i_g} \quad ? \vdash h : S^1_{i_h}}{? \vdash recint_1(f, k, x, app_1(g, app_1(f, app_1(h, x)))) : S^2_i} \quad i = max(i_g, i_h)$$

(Fonc)

$$? [f : s] \vdash f : s$$

D.5 Module d'inférence de schémas binaires

Les axiomes et règles d'inférence de schémas binaires sont les suivants :

(P_1)

$$\frac{? \vdash abs(f, x, e) : s_1 \quad ? [f : s_1] \vdash P : s_2}{? \vdash abs(f, x, e); P : s_2}$$

(P_2)

$$\frac{? \vdash recint_1(f, e_1, x, e_2) : s_1 \quad ? [f : s_1] \vdash P : s_2}{? \vdash recint_1(f, e_1, x, e_2); P : s_2}$$

(P_3)

$$\frac{? \vdash recint_2(f, x_1, e_1, x_2, x_3, e_2) : s_1 \quad ? [f : s_1] \vdash P : s_2}{? \vdash recint_2(f, x_1, e_1, x_2, x_3, e_2); P : s_2}$$

($Prim_1$)

$$? \vdash abs(f, x, k) : P_1$$

($Prim_2$)

$$? \vdash abs(f, x, app_2(plus, x, k)) : I_1$$

($Prim_2$')

$$\frac{k > 0}{? \vdash abs(f, x, app_2(plus, x, k)) : I'_1}$$

($Prim_2$")

$$\frac{k > 1}{? \vdash abs(f, x, app_2(plus, x, k)) : I''_1}$$

(Ssun-P)

$$\frac{? \vdash g : P_i}{? \vdash recint_1(f, k, x, app_1(g, app_1(f, x))) : P_{i+1}}$$

(Ssun-I)

$$\frac{? \vdash g : I'_i}{? \vdash recint_1(f, k, x, app_1(g, app_1(f, x))) : I_{i+1}} \quad k \geq i - 1$$

(Ssun-I'-1)

$$\frac{? \vdash g : I'_i}{? \vdash recint_1(f, k, x, app_1(g, app_1(f, x))) : I'_{i+1}} \quad k \neq 0$$

(Ssun-I'-2)

$$\frac{? \vdash g : I''_i}{? \vdash recint_1(f, 0, x, app_1(g, app_1(f, x))) : I'_{i+1}}$$

(Ssun-I"-1)

$$\frac{? \vdash g : I'_i}{? \vdash recint_1(f, k, x, app_1(g, app_1(f, x))) : I''_{i+1}} \quad k > 1$$

(Ssun-I"-2)

$$\frac{? \vdash g : I''_i}{? \vdash recint_1(f, 0, x, app_1(g, app_1(f, x))) : I''_{i+1}}$$

(Ssun-I"-3)

$$\frac{? \vdash g : I''_i}{? \vdash recint_1(f, 1, x, app_1(g, app_1(f, x))) : I''_{i+1}}$$

(Sbin_1)

$$\frac{? \vdash g : s_1 \quad ? \vdash h : s_2}{? \vdash recint_2(f, x_2, app_1(h, x_2), x_1, x_2, app_1(g, app_2(f, x_1, x_2))) : B^1(s_1, s_2)}$$

(Sbin_2)

$$\frac{? \vdash h : s_1 \quad ? \vdash g : s_2}{? \vdash recint_2(f, x_2, app_1(h, x_2), x_1, x_2, app_2(f, x_1, app_1(g, x_2))) : B^2(s_1, s_2)}$$

(Sbin_3)

$$\frac{? \vdash g : s}{? \vdash recint_2(f, x_2, k, x_1, x_2, app_2(g, x_2, app_2(f, x_1, x_2))) : B^3(s)}$$

(Fonc)

$$? [f : s] \vdash f : s$$

D.6 Module d'explication

Ce module sert à tracer le processus d'inférence de schémas. Cela permet de présenter à l'utilisateur, en plus des schémas calculés pour le programme et la propriété, le détail de l'inférence. Ainsi, on connaît précisément la succession des règles qui ont été utilisées, sur quelles fonctions elles ont été appliquées, et les schémas de toutes les fonctions intermédiaires.

Pour terminer, voici une copie d'écran de l'exécution du prototype sur l'exemple du programme *Reverse* et de la propriété Id :

```
#interface
   "letrec append case((nil,m:int), cons(m,nil),
                       (n:int::l:list, m:int), cons(n, append(l,m)));
    letrec reverse case(nil,nil,n:int::l:list,append(reverse(l),n))."
   "let id (x, x).";;
interface "letrec append case((nil,m:int), cons(m,nil), (n:int::l:list,
m:int), cons(n, append(l,m))); letrec reverse case(nil,nil,n:int::l:list,
append(reverse(l),n))." "let id (x, x).";;


PROGRAMME
Voici le texte d'entree "pretty_printe" :


letrec  append = (nil,m:int) -> cons(m,nil)
               (n:int::l:list,m:int) -> cons(n,append(l,m))

letrec  reverse = nil -> nil
                n:int::l:list -> append(reverse(l),n)


Voici maintenant son abstraction :


letrec  append = 0 -> +(0,1)
               +(l,1) -> +(append(l),1)
letrec  reverse = 0 -> 0
                +(l,1) -> append(reverse(l))


Ce qui donne apres normalisation :


letrec  append = 0 -> 1
               +(l,1) -> +(append(l),1)
letrec  reverse = 0 -> 0
                +(l,1) -> append(reverse(l))


D'ou la deduction du schema, avec une petite explication :

reverse est de schema 1S3 et est deduit de append (SSUN)
append est de schema 1S2 et est deduit de fonction 1S1 (SSUN)
fonction 1S1 est de schema 1S1 car elle provient d'une abstraction (PRIM)
```

```
PROPRIETE
Voici le texte d'entree "pretty_printe" :


Let id = x -> x


Ce qui donne apres normalisation :


Let id = x -> x


D'ou la deduction du schema, avec une petite explication :

id est de schema 1S1 car elle provient d'une abstraction (PRIM)



SCHEMA COMMUN au programme et a la propriete

Le schema unaire commun est : 1S3.

D'ou le jeu de test abstrait : {0,1,2,3}.

Un jeu de test concret possible est : {[],0::[],1::2::[],3::4::5::[]}.

- : unit = ()
#
```

Bibliographie

[ABR96] ABRIAL (J.-R.). – *The B-Book: Assigning Programs to Meanings.* – Cambridge University Press, 1996.

[ADH+89] AGRAWAL (H.), DEMILLO (R.), HATHAWAY (R.), HSU (W.), KRAUSER (E.), MARTIN (R.J.), MATHUR (A.) et SPAFFORD (E.). – Design of Mutant Operators for the C Programming Language. *Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana*, Mars 1989.

[BEI90] BEIZER (B.). – *Software Testing Techniques, 2^{nd} Edition.* – Van Nostrand Reinhold, 1990.

[BGM91] BERNOT (G.), GAUDEL (M-C.) et MARRE (B.). – Software Testing Based on Formal Specifications: a Theory and a Tool. *Software Engineering Journal*, Novembre 1991, pp. 387–405.

[BS87] BASILI (V.R.) et SELBY (R.W.). – Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, vol. 13, n° 12, Décembre 1987, pp. 1278–1296.

[BUD81] BUDD (T.A.). – Mutation Analysis: Ideas, Examples, Problems and Prospects. *Computer Program Testing*, 1981, pp. 129–148.

[CAR81] CARTWRIGHT (R.). – Formal Program Testing. *POPL*, Janvier 1981, pp. 125–132.

[CC77] COUSOT (P.) et COUSOT (R.). – Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL*, 1977, pp. 238–252.

[CC92] COUSOT (P.) et COUSOT (R.). – Abstract interpretation and application to logic programs. *J. Logic Programming*, vol. 13, n° 2-3, 1992, pp. 103–179.

[CM90] COBB (R.) et MILLS (H.). – Engineering Software under Statistical Quality Control. *IEEE Software*, Novembre 1990, pp. 44–54.

[CPRZ89] CLARKE (L.A.), PODGURSKY (A.), RICHARDSON (D.J.) et ZEIL (S.J.). – A Formal Evaluation of Data Flow Path Coverage Criteria. *IEEE Transactions on Software Engineering*, vol. 15, n° 11, Juin 1989, pp. 1318–1332.

[DAC] DWYER (M.B.), AVRUNIN (G.S.) et CORBETT (J.C.). – Property Specification Patterns for Finite-State Verification. *Technical Report KSU CIS TR-98-9.*

[DAC98] DWYER (M.B.), AVRUNIN (G.S.) et CORBETT (J.C.). – Property Specification Patterns for Finite-State Verification. *2^{nd} Workshop on Formal Methods in Software Practice*, Mars 1998.

[DF93] DICK (J.) et FAIVRE (A.). – Automating the generation and sequencing of test case from model-based specifications. *FME'93: Industrial Strength Formal Methods*, 1993.

[DKM+89] DEMILLO (R.A.), KRAUSER (E.W.), MARTIN (R.J.), OFFUTT (A.J.) et SPAFFORD (E.H.). – The Mothra Tool Set. *22^{nd} Hawaii International Conference on System Sciences*, Janvier 1989, pp. 275–284.

[DLS78] DEMILLO (R.A.), LIPTON (R.J.) et SAYWARD (F.G.). – Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, vol. 11, Avril 1978, pp. 34–41.

[DM95] DELAMARO (M.E.) et MALDONADO (J.C.). – Proteum: A Mutation Testing Tool for C Programs. *Purdue SERC Newsletter*, 1995.

[DN84] DURAN (J.W.) et NTAFOS (S.). – An evaluation of random testing. *IEEE Transactions on Software Engineering*, vol. SE-10, n° 4, Juillet 1984, pp. 438–444.

[DO91] DEMILLO (R.A.) et OFFUTT (A.J.). – Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, vol. 17, n° 9, Septembre 1991, pp. 900–910.

[FW88] FRANKL (P.G.) et WEYUKER (E.J.). – An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, vol. SE-14, n° 10, Octobre 1988, pp. 1483–1498.

[FW93a] FRANKL (P.G.) et WEISS (S.N.). – An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, vol. 19, n° 8, Août 1993, pp. 774–787.

[FW93b] FRANKL (P.G.) et WEYUKER (E.J.). – A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, vol. 19, n° 3, Mars 1993, pp. 202–213.

[FWW85] FRANKL (P.G.), WEISS (S.N.) et WEYUKER (E.J.). – ASSET - A system to select and evaluate tests. *IEEE Conf. Software Tools*, Avril 1985, pp. 72–79.

[GAU95] GAUDEL (M-C.). – Testing can be formal, too. *TAPSOFT*, 1995.

[GG75] GOODENOUGH (J.B.) et GERHART (S.L.). – Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, vol. 1, n° 2, Juin 1975, pp. 156–173.

[GH93] GUTTAG (J.V.) et HORNING (J.J.). – Larch: languages and tools for formal specification. *Texts and Monographs in Computer Science*, 1993.

[GMSB96] GAUDEL (M.-C.), MARRE (B.), SCHLIENGER (F.) et BERNOT (G.). – *Précis de génie logiciel.* – Masson, 1996.

[GOU83] GOURLAY (J.). – A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, vol. SE-9, n° 6, Novembre 1983, pp. 686–709.

[HAM77] HAMLET (R.G.). – Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, vol. 3, n° 4, Juillet 1977, pp. 279–290.

[HAM89] HAMLET (R.). – Theoretical comparison of testing methods. *SIGSOFT Symposium on Software Testing, Analysis, and Verification*, vol. 3, December 1989, pp. 28–37.

[HT90] HAMLET (D.) et TAYLOR (R.). – Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, vol. 16, Décembre 1990, pp. 206–215.

[JLT98] Jensen (T.), Le Métayer (D.) et Thorn (T.). – *Verification of control flow based security policies.* – Rapport technique n° 1210, IRISA, 1998.

[JON90] JONES (C.B.). – *Systematic software development using VDM.* – Prentice Hall International, second edition, 1990.

[KO91] KING (K.N.) et OFFUTT (A.J.). – A Fortran Language System for Mutation-based Software Testing. *Software-Practice and Experience*, vol. 21, n° 7, 1991, pp. 685–718.

[KOR90] KOREL (B.). – Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, vol. 16, n° 8, Août 1990, pp. 870–879.

[LMNR98] LE MÉTAYER (D.), NICOLAS (V.-A.) et RIDOUX (O.). – Exploring the Software Development Trilogy. – Novembre 1998. IEEE Software.

[NTA81] NTAFOS (S.C.). – On testing with required elements. *COMPSAC'81*, Novembre 1981, pp. 132–139.

[NTA88] NTAFOS (S.C.). – A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, vol. 14, n° 6, Juin 1988, pp. 868–874.

[OPV96] OFFUTT (A.J.), PAYNE (J.) et VOAS (J.M.). – Mutation Operators for Ada. *Technical Report ISSE-TR-96-06, Department of Information and Software Systems Engineering, George Mason University, Fairfax, Virginia*, Mars 1996.

[ORZ93] OFFUTT (A.J.), ROTHERMEL (G.) et ZAPF (C.). – An experimental evaluation of selective mutation. *ICSE-15*, Mai 1993, pp. 100–107.

[OW91] OSTRAND (T.J.) et WEYUKER (E.J.). – Data Flow-Based Test Adequacy Analysis for Languages with Pointers. *POPL*, Janvier 1991, pp. 100–125.

[PDM89] PIERCE (B.), DIETZEN (S.) et MICHAYLOV (S.). – Programming in Higher-Order Typed Lambda-Calculi. *CMU-CS-89-111*, 1989.

[PMW93] PAULIN-MOHRING (C.) et WERNER (B.). – Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, no15, 1993.

[RC85] RICHARDSON (D.J.) et CLARKE (L.A.). – Partition Analysis: A Method Combining Testing and Verification. *IEEE Transactions on Software Engineering*, vol. 11, n° 12, Décembre 1985, pp. 1477–1490.

[RW85] RAPPS (S.) et WEYUKER (E.J.). – Selecting Software Test Data Using Dataflow Information. *IEEE Transactions on Software Engineering*, vol. 11, n° 4, Avril 1985, pp. 367–375.

[SPI92] SPIVEY (M.). – *The Z notation - A reference manual.* – Prentice Hall International, second edition, 1992, *International Series in Computer Science.*

[TFWC91] THÉVENOD-FOSSE (P.), WAESELYNCK (H.) et CROUZET (Y.). – An Experimental Study on Software Structural Testing: Deterministic versus Random Input Generation. *21st IEEE International Symposium on Fault-Tolerant Computing (FTCS-21)*, 1991, pp. 410–417.

[VA98] VAN AERTRYCK (L.). – *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels.* – Thèse de PhD, Université de Rennes 1, Janvier 1998.

[VMM91] VOAS (J.M.), MORELL (L.) et MILLER (K.W.). – Predicting Where Faults Can Hide from Testing. *IEEE Software*, vol. 8, n° 2, 1991, pp. 41–58.

[WAE93] WAESELYNCK (H.). – *Vérification de logiciels critiques par le test statistique.* – Thèse de PhD, Institut national polytechnique de Toulouse, Janvier 1993.

[WO80] WEYUKER (E.J.) et OSTRAND (T.J.). – Theories of program testing and the application of revealing sub-domains. *IEEE Transactions on Software Engineering*, vol. SE-6, n° 3, Mai 1980, pp. 236–246.

[XMDA+94] XANTHAKIS (S.), MAURICE (M.), DE AMESCUA (A.), HOURI (O.) et GRIFFET (L.). – *Test & Contrôle des Logiciels : Méthodes, Techniques & Outils.* – EC2, 1994.

Résumé

Le problème abordé dans cette thèse concerne la production automatique de données de test permettant de prouver des propriétés de programmes. Nous nous situons ainsi à mi-chemin entre le domaine du test et celui de la vérification de programmes. Les travaux dans le domaine du test ont conduit à des outils semi-automatiques d'utilisation simple, mais qui reposent sur des hypothèses difficilement vérifiables en pratique. Dans le domaine de la vérification, des outils basés sur des méthodes formelles ont été développés, mais ils nécessitent un utilisateur expert dans les techniques de preuve utilisées par l'outil. Cette situation est due aux problèmes d'indécidabilité engendrés par la puissance des formalismes traités. La thèse que nous présentons est qu'il est possible de développer des méthodes formelles automatiques pour prouver des propriétés de programmes, à condition de considérer des formalismes restreints. Notre principale contribution est une nouvelle approche pour la vérification de programmes, intégrant les techniques de test et d'analyse statique.

Nous proposons une méthode formelle de génération de jeux de test finis complets permettant de prouver qu'un programme vérifie une propriété donnée. Cette méthode utilise le texte du programme et de la propriété, qui doivent appartenir à certaines classes de programmes (ou de propriétés). Ces classes sont représentées par des hiérarchies de schémas, qui peuvent être vues comme modélisant des hypothèses de test. Tout programme appartenant à un de nos schémas et passant le jeu de test avec succès vérifie la propriété testée. Pour une propriété donnée, notre méthode est complètement automatique et ne nécessite donc aucune compétence particulière de l'utilisateur. Nous avons implanté cette méthode dans un prototype (traitant un langage fonctionnel restreint), pour le cas de propriétés s'exprimant en termes de *longueur de liste*.

Mots clés

Génie logiciel, test structurel de programmes, génération automatique de jeux de test, vérification de programmes, analyse de programmes, schémas de programmes.